emUSB-Device

USB Device stack

User Guide & Reference Manual

Document: UM09001
Software Version: 3.78.0
Revision: 0
Date: November 20, 2026



A product of SEGGER Microcontroller GmbH

www.segger.com

Disclaimer

The information written in this document is assumed to be accurate without guarantee. The information in this manual is subject to change for functional or performance improvements without notice. SEGGER Microcontroller GmbH (SEGGER) assumes no responsibility for any errors or omissions in this document. SEGGER disclaims any warranties or conditions, express, implied or statutory for the fitness of the product for a particular purpose. It is your sole responsibility to evaluate the fitness of the product for any specific use.

Copyright notice

You may not extract portions of this manual or modify the PDF file in any way without the prior written permission of SEGGER. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

© 2010-2026 SEGGER Microcontroller GmbH, Monheim am Rhein / Germany

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH

Ecolab-Allee 5
D-40789 Monheim am Rhein

Germany

Tel. +49 2173-99312-0
Fax. +49 2173-99312-28
E-mail: ticket_emusb@segger.com*
Internet: www.segger.com

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

Manual versions

This manual describes the current software version. If you find an error in the manual or a problem in the software, please inform us, and we will try to assist you as soon as possible. Contact us for further information on topics or functions that are not yet documented.

As of version 3.00 the history has been reset. Older history entries can be found in older versions of this document.

Print date: November 20, 2026

Software	Date	By	Description
3.78.0	2026-11-20	RH	Corrected typos. Added section *Security considerations*.
3.76.0	2026-09-23	SR	Several typos were corrected.
3.74.0	2026-08-26	RH	Allow dynamic allocation of the descriptor buffer using `USBD_X_GetDescBuffer()`. Updated section *PSOC driver*. New section *System architecture*.
3.72.0	2026-07-13	RH	Added section *Atmel / Microchip high-speed driver*. Function `USBD_SetOnRxEP0()` replaced by `USBD_SetOnRxEP0Hook()`. Improved sections *USB suspend* and *USB disconnect*.
3.70.0	2025-11-24	RH	Added function *`USBD_BULK_ReadRecord()`*. Added function *`USB_DRIVER_Infineon_PSOCE84_DMA_SetPLLRefClock()`*.
3.68.0	2025-08-28	RH	Update to latest software version.
3.66.3	2025-03-24	RH	Added function *`USBD_MTP_SetWriteProtectionCb()`*.
3.66.0	2024-10-02	RH	Added section *PSOCE84 driver*.
3.64.5	2024-09-11	RH	Add write protection for MTP storage: *`USB_MTP_INST_DATA_DRIVER`*
3.64.3	2024-06-07	RH	Update to latest software version.
3.64.2	2024-05-16	RH	Update to latest software version.
3.64.1	2024-04-09	RH	Updated section *PSoC6 driver*.
3.64.0	2024-02-13	RH	Updated section *Compile-time configuration*. Added section *C++ Host API*.
3.62.0	2023-11-21	RH	Update to latest software version.
3.60.4	2023-10-18	RH	Update to latest software version.
3.60.2	2023-07-10	YR	Update to latest software version.
3.60.1	2023-06-30	RH	Update to latest software version.
3.60.0	2023-04-12	RH	Link Power Management (LPM) • Added function `USBD_SetLPMResponse()`.
3.58.0	2023-02-28	YR	MTP class: • Added function `USBD_MTP_SetOperationCb()`.
3.56.0	2023-01-12	RH	Link Power Management (LPM) • Added function `USBD_GetDeviceState()`. • Added function `USBD_SetBESLValues()`. • Added function `USBD_SetOnLPMChange()`. MTP class: • Added function `USBD_MTP_RemoveStorage()`.
3.54.0	2022-12-07	RH	Update to latest software version.
3.52.2	2022-10-25	RH	Added functions `USB_DRIVER_Cypress_PSoC6_SysTick()` and `USB_DRIVER_Cypress_PSoC6_Resume()`.
3.52.1	2022-08-24	SR	Updated section *PSoC6 driver*.
3.52.0	2022-08-15	RH	Added section *Link Power Management (LPM)*. HID class: • Added functions `USBD_HID_Receive()` and `USBD_HID_ReceivePoll()`.
3.50.2	2022-06-23	RH	Updated section *Giga Device GD32F4xx driver (high-speed controller)*.
3.50.1	2022-06-01	RH	Updated section *Giga Device GD32F4xx driver (full-speed controller)*.
3.50.0	2022-05-13	RH	BULK and VSC class • Update section *Example Application*. *Communication Device Class (CDC)*

 © 2010-2026 SEGGER Microcontroller GmbH

Software	Date	By	Description
			• Updated driver requirements for Host system Updated section *Synopsys DWC2 driver (DMA mode)*
3.46.4	2022-03-14	SR	Core functions: • Added function `USBD_SetOnSOF()`. Getting Started: • Wrong path to emUSB-Device was used (USBD instead of USB).
3.46.3	2022-02-22	YR	MTP class: • Added function `USBD_MTP_SetObjectAllocFailCb()`.
3.46.0	2021-10-15	RH	Added section *Low power mode*.
3.44.0	2021-09-17	RH	Added description for `USB_ON_CLASS_REQUEST` and `USB_ON_SETUP`.
3.42.1	2021-07-29	RH	Added section *Giga Device GD32F450 driver*.
3.42.0	2021-07-12	RH	HID class: • Added function `USBD_HID_AddEx()`.
3.40.0	2021-03-31	RH	Add new chapter *Vendor Specific Class (VSC)*.
3.38.0	2021-01-29	RH	DFU class: • Added function `USBD_DFU_AddAlternateInterface()`. Printer class: • Added function `USB_PRINTER_SetClass()`.
3.36.3	2020-12-18	YR	Update to latest software version.
3.36.2	2020-12-15	RH	BULK class: • Added function `USBD_BULK_PollForTX()`. CDC class: • Added functions `USBD_CDC_PollForRX()`, `USBD_CDC_PollForTX()` and `USBD_CDC_ReceivePoll()`. Added new section *Timeout handling* Update to latest software version.
3.36.1	2020-10-30	YR	Update to latest software version.
3.36.0	2020-10-12	YR	Audio class: • Updated API with new function and structure names. Update to latest software version.
3.34.3	2020-08-25	YR	Update to latest software version.
3.34.2	2020-07-23	YR	Update to latest software version.
3.34.1	2020-06-10	RH	Added section *Synopsys DWC2 driver (DMA mode)*
3.34.0	2020-03-20	RH	BULK class: • Added function `USBD_BULK_AddAlternateInterface()`.
3.32.0	2020-02-26	RH	Added CCID chapter.
3.30.0	2020-01-27	YR	Added MIDI chapter.
3.28.1	2019-12-13	YR	Update to latest software version. HID class: Added `USBHID_Read()` and `USBHID_Write()` functions.
3.28.0	2019-11-29	YR	Printer class: • Added function `USB_PRINTER_SetOnVendorRequest()`.
3.26	2019-08-30	RH	USB Core: • Added function `USBD_GetVersion()`. • Added function `USBD_SetCheckAddress()`. UVC class: • Added function `USBD_UVC_WriteEx()`.
3.24	2019-07-01	RH	Added chapter *USB Device Firmware Upgrade (DFU)*
3.22	2019-06-17	RH	BULK class: • Added function `USBD_BULK_PollForRX()`. • Added function `USBD_BULK_ReceivePoll()`. USB Core: • Added function `USBD_WaitForEndOfTransferEx()`. • Added function `USBD_ReceivePoll()`. • Added function `USBD_EnableSuperSpeed()`. • Added function `USBD_SetWebUSBInfo()`. MSD class: • Added function `USBD_MSD_Poll()`. Add section *XHCI driver* Add chapter *USB Video device Class (UVC)*

 © 2010-2026 SEGGER Microcontroller GmbH

Software	Date	By	Description
3.20	2019-05-15	YR	Update to latest software version. SmartMSD was renamed to VirtualMSD. Function prefixes changed from "SMSD" to "VMSD". HID class: Added `USBD_HID_ReadReport()` function.
3.18b	2019-03-04	YR	Update to latest software version.
3.18a	2019-02-28	YR	Update to latest software version. Bulk Host API: • Removed `USBBULK_ResetPipe()` • Added `USBBULK_ResetINPipe()` • Added `USBBULK_ResetOUTPipe()` • `USBBULK_DEV_INFO` received an additional member - "Speed"
3.18	2018-11-26	RH	Update to latest software version.
3.16	2018-10-05	YR	Update to latest software version. MTP class: • `USBD_MTP_SendEvent()` description updated.
3.14	2018-07-19	RH	Added section *Device driver specifics* on page 751 BULK class: • Added functions `USBD_BULK_ReadAsync()` and `USBD_BULK_WriteAsync()`. CDC class: • Added functions `USBD_CDC_ReadAsync()` and `USBD_CDC_WriteAsync()`.
3.12	2018-05-04	RH	USB Core: • Added function `USBD_RemoveOnEvent()`. • Removed functions `USBD_SetLogFunc()` and `USBD_SetWarnFunc()`. Audio class: • Added function `USBD_AUDIO_Read_Task()`. • Added function `USBD_AUDIO_Write_Task()`.
3.10	2018-03-22	RH	BULK class: • Added function `USBD_BULK_Add_Ex()`. Update to latest software version. Added Audio chapter.
3.08	2018-02-12	RH	USB Core: • Added function `USBD_RegisterSCHook()`. • Added function `USBD_AddEPEx()`.
3.06e	2018-01-12	RH	Update to latest software version.
3.06d	2017-12-19	YR	Update to latest software version.
3.06c	2017-12-04	RH	USB Core: • Added I/O functions.
3.06b	2017-10-13	YR	Update to latest software version. Corrected `USBD_WriteEP0FromISR` name (was `USB__WriteEP0FromISR`).
3.06	2017-09-15	RH	Printer class: • Added function `USB_PRINTER_ConfigIRQProcessing()`. • Added function `USB_PRINTER_TaskEx()`. USB Core: • Added `USBD_SetCacheConfig()`. Chapter "Getting started" revised.
3.04	2017-07-24	YR	Update to latest software version. Added chapter "emUSB-Device-IP"
3.02q	2017-07-17	YR	Update to latest software version.
3.02p	2017-07-14	RH	Update to latest software version.
3.02o	2017-07-10	YR	Chapter Combining USB components: • Added information on the MSD+MTP combination feature. Added Chapter "Profiling with SystemView".
3.02o	2017-07-01	RH	Major revision of the manual. • Manual converted to text processor emDoc.
3.02n	2017-06-12	SR	Update to latest software version.
3.02m	2017-06-08	RH	Update to latest software version.
3.02l	2017-06-02	SR	Update to latest software version.
3.02k	2017-04-10	RH	Function `USBD_AddEP()`: Parameter 'Interval' changed.
3.02j	2017-03-01	RH	Update to latest software version.

 © 2010-2026 SEGGER Microcontroller GmbH

Software	Date	By	Description
3.02i	2017-01-26	SR	Update to latest software version.
3.02h	2017-01-26	SR	Update to latest software version.
3.02g	2017-01-23	SR	Update to latest software version.
3.02f	2017-01-04	RH	Update to latest software version.
3.02e	2016-12-15	RH	Update to latest software version.
3.02d	2016-11-18	RH	Update to latest software version.
3.02c	2016-11-03	RH	Update to latest software version.
3.02b	2016-10-28	RH	Chapter VirtualMSD: • Renamed functions according to the emUSB V3 conventions.
3.02a	2016-10-18	RH	Update to latest software version.
3.02	2016-10-07	SR	Chapter CDC: • Updated Overview - Added Windows 10 Support. • Updated Installing the driver. Section verification combined with Installing driver. • Updated section The .inf file • Added new section: Signing the package • Testing communication to the USB device updated.
3.02	2016-09-30	RH	Chapter CDC Data structures: • Removed CTS from `USB_CDC_SERIAL_STATE`. Chapter BULK communication: • Removed description of the Segger USB driver (not necessary anymore) • BULK host API and sample applications support for Linux and MacOSX added. • Added new function `USBD_BULK_SetMSDescInfo()`. • Added new function `USBBULK_GetDevInfoByIdx()`. • Removed description of deprecated host API functions. Chapter VirtualMSD: • Added memory usage calculation. Many minor corrections.
3.00g	2016-08-22	RH	Update to latest software version. Chapter Target OS Interface: • New advanced OS layer interface. Chapter Mass Storage Device Class (MSD): • Added new function `USBD_MSD_RequestRefresh()`
3.00f	2016-07-20	YR	Update to latest software version. Chapter VirtualMSD: • Changed function prefix to VMSD. • Removed obsolete functions.
3.00e	2016-07-08	RH	Update to latest software version.
3.00d	2016-06-08	YR	Update to latest software version.
3.00c	2016-05-23	YR	Update to latest software version. Chapter Bulk communication: • Added paragraph “Writing your own host driver”.
3.00b	2016-04-27	YR	Update to latest software version.
3.00a	2016-04-15	SR	Chapter USB Core functions: • Updated prototype for `USBD_SetMaxPower`. Chapter HID: • Added new function for setting a callback for `SET_REPORT`. Chapter Debugging: • Changed all prototypes from `USB_*` to `USBD_*`.
3.00	2016-02-12	YR	Initial Version

 © 2010-2026 SEGGER Microcontroller GmbH

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (compiler, linker, Integrated Development Environment).
- The C programming language.
- The target processor.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command prompt or that appears on the display (that is, system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables, and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

 © 2010-2026 SEGGER Microcontroller GmbH

 © 2010-2026 SEGGER Microcontroller GmbH

Table of contents

1 Introduction 27
1.1 Overview 28
1.2 emUSB-Device features 29
1.3 System architecture 30
1.3.1 External components interfacing to emUSB-Device 30
1.3.1.1 Application 30
1.3.1.2 File system 30
1.3.1.3 Storage module 30
1.3.1.4 TCP/IP stack 31
1.3.1.5 RTOS 31
1.3.1.6 USB controller hardware 31
1.4 emUSB-Device components 32
1.4.1 emUSB-Device-Bulk 32
1.4.2 emUSB-Device-MSD 32
1.4.2.1 Purpose of emUSB-Device-MSD 32
1.4.2.2 Typical applications 33
1.4.2.3 emUSB-Device-MSD features 33
1.4.2.4 How does it work? 33
1.4.3 emUSB-Device IP-over-USB 34
1.4.3.1 Typical applications 34
1.4.4 emUSB-Device-VirtualMSD 34
1.4.4.1 Typical applications 34
1.4.5 emUSB-Device-CDC 34
1.4.5.1 Typical applications 34
1.4.6 emUSB-Device-HID 34
1.4.6.1 Typical applications 34
1.4.7 emUSB-Device-MTP 35
1.4.7.1 Typical applications 35
1.4.8 emUSB-Device-Printer 35
1.4.8.1 Typical applications 35
1.4.9 emUSB-Device-RNDIS 35
1.4.9.1 Typical applications 35
1.4.10 emUSB-Device-CDC-ECM 36
1.4.10.1 Typical applications 36
1.4.11 OS Layer 36
1.5 Requirements 37
1.5.1 Target system 37
1.5.2 Development environment (compiler) 37
1.6 File structure 38
1.7 Multithreading 39

 © 2010-2026 SEGGER Microcontroller GmbH

2 Background information 40
2.1 USB 41
2.1.1 Short Overview 41
2.1.2 Important USB standard versions 41
2.1.3 USB system architecture 42
2.1.4 Transfer types 44
2.1.5 Setup phase / enumeration 44
2.1.6 Product / Vendor IDs 44
2.2 Predefined device classes 45
2.3 USB hardware analyzers 46
2.4 References 47

3 Getting started 48
3.1 How to set up your target system 49
3.1.1 Take a running project 49
3.1.2 Add emUSB-Device files 49
3.1.3 Configuring debugging output 49
3.1.4 Add hardware dependent configuration 50
3.1.5 Prepare and run the application 50
3.2 Updating emUSB-Device 52
3.3 emUSB-Device Configuration 53
3.3.1 USB_DEVICE_INFO 53
3.3.2 Additional required configuration for emUSB-MSD 54
3.3.3 Descriptors 54
3.4 Compile-time configuration 56
3.4.1 Compile-time switches for debugging 56
3.4.1.1 USB_DEBUG_LEVEL 56
3.4.1.2 USB_LOG_BUFFER_SIZE 56
3.4.2 Use of standard C-library functions 56
3.4.3 General USB configuration 57
3.4.3.1 USB_SUPPORT_TRANSFER_ISO 57
3.4.3.2 USB_SUPPORT_TEST_MODE 57
3.4.3.3 USB_NUM_EPS 57
3.4.3.4 USB_MAX_NUM_IF 57
3.4.3.5 USB_MAX_NUM_ALT_IF 57
3.4.3.6 USB_DESC_BUFFER_SIZE 58
3.4.3.7 USBD_SUPPORT_PROFILE 58
3.4.3.8 USBD_OS_USE_USBD_X_INTERRUPT 58
3.4.3.9 USBD_OS_USE_ISR_FLAG 59
3.5 Security considerations 60
3.6 Host OS specifics 61
3.6.1 Windows registry 61
3.6.1.1 Cleaning the Windows registry 61

4 USB Core 62
4.1 Overview 63
4.2 Target API 64
4.2.1 USB basic functions 67
4.2.1.1 USBD_GetState() 67
4.2.1.2 USBD_GetSpeed() 68
4.2.1.3 USBD_GetDeviceState() 69
4.2.1.4 USBD_Init() 70
4.2.1.5 USBD_IsConfigured() 71
4.2.1.6 USBD_Start() 72
4.2.1.7 USBD_Stop() 73
4.2.1.8 USBD_DeInit() 74
4.2.1.9 USBD_GetVersion() 75
4.2.2 USB configuration functions 76

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.1 USBD_AddDriver() 76
4.2.2.2 USBD_SetISREnableFunc() 77
4.2.2.3 USBD_SetAttachFunc() 78
4.2.2.4 USBD_AddEP() 79
4.2.2.5 USBD_AddEPEx() 80
4.2.2.6 USBD_SetDeviceInfo() 81
4.2.2.7 USBD_SetClassRequestHook() 82
4.2.2.8 USBD_SetVendorRequestHook() 83
4.2.2.9 USBD_SetIsSelfPowered() 84
4.2.2.10 USBD_SetMaxPower() 85
4.2.2.11 USBD_SetOnEvent() 86
4.2.2.12 USBD_RemoveOnEvent() 88
4.2.2.13 USBD_SetOnRxEP0Hook() 89
4.2.2.14 USBD_SetOnRXHookEP() 90
4.2.2.15 USBD_SetOnSetup() 91
4.2.2.16 USBD_SetOnSetupHook() 92
4.2.2.17 USBD_SetOnSOF() 93
4.2.2.18 USBD_RemoveOnSOF() 94
4.2.2.19 USBD_WriteEP0FromISR() 95
4.2.2.20 USBD_EnableIAD() 96
4.2.2.21 USBD_SetCacheConfig() 97
4.2.2.22 USBD_RegisterSCHook() 98
4.2.2.23 USBD_AssignMemory() 99
4.2.2.24 USBD_UseV210() 100
4.2.2.25 USBD_SetBESLValues() 101
4.2.2.26 USBD_SetOnLPMChange() 102
4.2.2.27 USBD_SetLPMResponse() 103
4.2.2.28 USBD_EnableSuperSpeed() 104
4.2.2.29 USBD_SetWebUSBInfo() 105
4.2.2.30 USBD_SetCheckAddress() 106
4.2.2.31 USBD_SetGetStringHook() 107
4.2.2.32 USBD_X_GetDescBuffer() 108
4.2.3 USB I/O functions 109
4.2.3.1 USBD_Read() 109
4.2.3.2 USBD_ReadOverlapped() 110
4.2.3.3 USBD_Receive() 111
4.2.3.4 USBD_ReceivePoll() 112
4.2.3.5 USBD_ReadAsync() 113
4.2.3.6 USBD_Write() 114
4.2.3.7 USBD_WriteAsync() 115
4.2.3.8 USBD_CancelIO() 116
4.2.3.9 USBD_WaitForEndOfTransferEx() 117
4.2.3.10 USBD_WaitForTXReady() 118
4.2.3.11 USBD_GetNumBytesInBuffer() 119
4.2.3.12 USBD_GetNumBytesRemToRead() 120
4.2.3.13 USBD_GetNumBytesRemToWrite() 121
4.2.3.14 USBD_StallEP() 122
4.2.4 USB Remote wakeup functions 123
4.2.4.1 USBD_SetAllowRemoteWakeUp() 124
4.2.4.2 USBD_DoRemoteWakeup() 125
4.2.5 Data structures 126
4.2.5.1 USB_ADD_EP_INFO 126
4.2.5.2 USB_SETUP_PACKET 127
4.2.5.3 SEGGER_CACHE_CONFIG 128
4.2.5.4 USB_CHECK_ADDRESS_FUNC 129
4.2.5.5 USB_ASYNC_IO_CONTEXT 130
4.2.5.6 USB_WEBUSB_INFO 131
4.2.6 Function Types 132
4.2.6.1 USB_ON_CLASS_REQUEST 132
4.2.6.2 USB_ON_SETUP 133

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.6.3 USB_GET_STRING_FUNC 134
4.2.6.4 USB_ON_LPM_CHANGE 135
4.3 Timeout handling 136
4.4 USB disconnect 137
4.5 Low power mode 138
4.5.1 USB suspend 138
4.5.2 Link Power Management (LPM) 139

5 Bulk communication 141

5.1 Generic bulk stack 142
5.2 Requirements for the Host (PC) 143
5.2.1 Windows 143
5.2.2 Linux 143
5.2.3 macOS 143
5.3 Example application 144
5.3.1 Running the example applications 144
5.3.2 Compiling the PC example application 145
5.3.2.1 Windows 145
5.3.2.2 Linux 146
5.3.2.3 macOS 146
5.4 Target API 147
5.4.1 Target interface function list 147
5.4.2 USB-Bulk functions 149
5.4.2.1 USBD_BULK_Add() 149
5.4.2.2 USBD_BULK_Add_Ex() 150
5.4.2.3 USBD_BULK_AddAlternateInterface() 151
5.4.2.4 USBD_BULK_SetMSDescInfo() 152
5.4.2.5 USBD_BULK_CancelRead() 153
5.4.2.6 USBD_BULK_CancelWrite() 154
5.4.2.7 USBD_BULK_GetNumBytesInBuffer() 155
5.4.2.8 USBD_BULK_GetNumBytesRemToRead() 156
5.4.2.9 USBD_BULK_GetNumBytesRemToWrite() 157
5.4.2.10 USBD_BULK_Read() 158
5.4.2.11 USBD_BULK_ReadAsync() 159
5.4.2.12 USBD_BULK_ReadOverlapped() 160
5.4.2.13 USBD_BULK_Receive() 161
5.4.2.14 USBD_BULK_ReadRecord() 162
5.4.2.15 USBD_BULK_ReceivePoll() 163
5.4.2.16 USBD_BULK_SetContinuousReadMode() 164
5.4.2.17 USBD_BULK_SetOnSetupRequest() 165
5.4.2.18 USBD_BULK_SetOnRXEvent() 166
5.4.2.19 USBD_BULK_SetOnTXEvent() 168
5.4.2.20 USBD_BULK_TxIsPending() 170
5.4.2.21 USBD_BULK_WaitForRX() 171
5.4.2.22 USBD_BULK_PollForRX() 172
5.4.2.23 USBD_BULK_WaitForTX() 173
5.4.2.24 USBD_BULK_PollForTX() 174
5.4.2.25 USBD_BULK_WaitForTXReady() 175
5.4.2.26 USBD_BULK_Write() 176
5.4.2.27 USBD_BULK_WriteAsync() 178
5.4.2.28 USBD_BULK_WriteEx() 179
5.4.3 Data structures 180
5.4.3.1 USB_BULK_INIT_DATA 180
5.4.3.2 USB_BULK_INIT_DATA_EX 181
5.5 C Host API 182
5.5.1 Bulk Host API list 183
5.5.2 USB-Bulk basic functions 185
5.5.2.1 USBBULK_Init() 185
5.5.2.2 USBBULK_Exit() 186

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.2.3 USBBULK_AddAllowedDeviceItem() 187
5.5.2.4 USBBULK_GetNumAvailableDevices() 188
5.5.2.5 USBBULK_Open() 189
5.5.2.6 USBBULK_Close() 190
5.5.3 USB-Bulk direct input/output functions 191
5.5.3.1 USBBULK_Read() 191
5.5.3.2 USBBULK_ReadTimed() 192
5.5.3.3 USBBULK_Write() 193
5.5.3.4 USBBULK_WriteTimed() 194
5.5.3.5 USBBULK_CancelRead() 195
5.5.3.6 USBBULK_FlushRx() 196
5.5.4 USB-Bulk control functions 197
5.5.4.1 USBBULK_SetMode() 197
5.5.4.2 USBBULK_GetMode() 198
5.5.4.3 USBBULK_SetReadTimeout() 199
5.5.4.4 USBBULK_SetWriteTimeout() 200
5.5.4.5 USBBULK_ResetINPipe() 201
5.5.4.6 USBBULK_ResetOUTPipe() 202
5.5.4.7 USBBULK_ResetDevice() 203
5.5.5 USB-Bulk general GET functions 204
5.5.5.1 USBBULK_GetVersion() 204
5.5.5.2 USBBULK_GetDevInfo() 205
5.5.5.3 USBBULK_GetDevInfoByIdx() 206
5.5.5.4 USBBULK_GetUSBId() 207
5.5.5.5 USBBULK_GetProductName() 208
5.5.5.6 USBBULK_GetVendorName() 209
5.5.5.7 USBBULK_GetSN() 210
5.5.5.8 USBBULK_GetConfigDescriptor() 211
5.5.6 USB-Bulk data structures 212
5.5.6.1 USBBULK_DEV_INFO 212

6 Vendor Specific Class (VSC) 213

6.1 Vendor Specific Class 214
6.2 Requirements for the Host (PC) 215
6.2.1 Windows 215
6.2.2 Linux 215
6.2.3 macOS 215
6.3 Example application 216
6.3.1 Running the example applications 216
6.3.2 Compiling the PC example application 217
6.3.2.1 Windows 217
6.3.2.2 Linux 217
6.3.2.3 macOS 217
6.4 Target API 218
6.4.1 Target interface function list 218
6.4.2 USB-VSC functions 220
6.4.2.1 USBD_VSC_Add() 220
6.4.2.2 USBD_VSC_AddAlternateInterface() 221
6.4.2.3 USBD_VSC_CancelIO() 222
6.4.2.4 USBD_VSC_GetNumBytesInBuffer() 223
6.4.2.5 USBD_VSC_GetNumBytesRemToRead() 224
6.4.2.6 USBD_VSC_GetNumBytesRemToWrite() 225
6.4.2.7 USBD_VSC_Read() 226
6.4.2.8 USBD_VSC_ReadAsync() 228
6.4.2.9 USBD_VSC_SetContinuousReadMode() 229
6.4.2.10 USBD_VSC_SetOnSetupRequest() 230
6.4.2.11 USBD_VSC_SetOnEPEvent() 231
6.4.2.12 USBD_VSC_TxIsPending() 235
6.4.2.13 USBD_VSC_WaitEP() 236

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.14 USBD_VSC_TxIsPending() 237
6.4.2.15 USBD_VSC_WaitEP() 238
6.4.2.16 USBD_VSC_PollEP() 239
6.4.2.17 USBD_VSC_WaitForTXReady() 241
6.4.2.18 USBD_VSC_Write() 242
6.4.2.19 USBD_VSC_WriteAsync() 244
6.4.3 Data structures 245
6.4.3.1 USB_VSC_INIT_DATA 245
6.4.3.2 USB_VSC_MSOSDESC_INFO 246
6.4.3.3 USB_VSC_ON_ADD_FUNCTION_DESC 247
6.4.3.4 USB_VSC_ON_SET_INTERFACE 248
6.5 C++ Host API 249

7 Mass Storage Device Class (MSD) 250

7.1 Overview 251
7.2 MSD Configuration 252
7.2.1 Initial configuration 252
7.2.2 Final configuration 252
7.2.3 MSD class specific configuration functions 252
7.2.4 Running the example application 252
7.2.4.1 MSD_Start_StorageRAM.c in detail 253
7.3 Target API 254
7.3.1 API functions 256
7.3.1.1 USBD_MSD_Add() 256
7.3.1.2 USBD_MSD_AddUnit() 257
7.3.1.3 USBD_MSD_AddCDRom() 258
7.3.1.4 USBD_MSD_SetPreventAllowRemovalHook() 259
7.3.1.5 USBD_MSD_SetReadWriteHook() 260
7.3.1.6 USBD_MSD_Task() 261
7.3.1.7 USBD_MSD_Poll() 262
7.3.1.8 USBD_MSD_PollEx() 263
7.3.1.9 USBD_MSD_SetStartStopUnitHook() 264
7.3.2 Extended API functions 265
7.3.2.1 USBD_MSD_Connect() 265
7.3.2.2 USBD_MSD_Disconnect() 266
7.3.2.3 USBD_MSD_RequestDisconnect() 267
7.3.2.4 USBD_MSD_RequestRefresh() 268
7.3.2.5 USBD_MSD_UpdateWriteProtect() 269
7.3.2.6 USBD_MSD_WaitForDisconnection() 270
7.3.3 Data structures 271
7.3.3.1 USB_MSD_INIT_DATA 271
7.3.3.2 USB_MSD_INFO 272
7.3.3.3 USB_MSD_INST_DATA 273
7.3.3.4 USB_MSD_LUN_INFO 274
7.3.3.5 PREVENT_ALLOW_REMOVAL_HOOK 275
7.3.3.6 READ_WRITE_HOOK 276
7.3.3.7 USB_MSD_INST_DATA_DRIVER 277
7.3.3.8 USB_MSD_STORAGE_API 279
7.3.3.9 START_STOP_UNIT_HOOK 280
7.4 MSD Storage Driver 281
7.4.1 General information 281
7.4.1.1 Supported storage types 281
7.4.1.2 Storage drivers supplied with this release 281
7.4.2 Interface function list 281
7.4.3 USB_MSD_STORAGE_API in detail 282
7.4.3.1 USB_MSD_STORAGE_INIT 282
7.4.3.2 USB_MSD_STORAGE_GETINFO 283
7.4.3.3 USB_MSD_STORAGE_GETREADBUFFER 284
7.4.3.4 USB_MSD_STORAGE_READ 285

 © 2010-2026 SEGGER Microcontroller GmbH

7.4.3.5 USB_MSD_STORAGE_GETWRITEBUFFER 286
7.4.3.6 USB_MSD_STORAGE_WRITE 287
7.4.3.7 USB_MSD_STORAGE_MEDIUMISPRESENT 288
7.4.3.8 USB_MSD_STORAGE_DEINIT 289

8 Virtual Mass Storage Component (VirtualMSD) 290

8.1 Overview 291
8.2 Configuration 292
8.2.1 Initial configuration 292
8.2.2 Final configuration 292
8.2.3 Class specific configuration functions 292
8.2.3.1 USB_VMSD_X_Config() 293
8.2.4 Running the example application 293
8.2.5 Calculation of RAM memory usage for VirtualMSD 294
8.3 Target API 296
8.3.1 API functions 297
8.3.1.1 USBD_VMSD_Add() 297
8.3.1.2 USB_VMSD_X_Config() 298
8.3.1.3 USBD_VMSD_AssignMemory() 299
8.3.1.4 USBD_VMSD_SetUserAPI() 300
8.3.1.5 USBD_VMSD_SetNumRootDirSectors() 301
8.3.1.6 USBD_VMSD_SetVolumeInfo() 302
8.3.1.7 USBD_VMSD_AddConstFiles() 303
8.3.1.8 USBD_VMSD_SetNumSectors() 304
8.3.1.9 USBD_VMSD_SetSectorsPerCluster() 305
8.3.2 Data structures 306
8.3.2.1 USB_VMSD_CONST_FILE 306
8.3.2.2 USB_VMSD_USER_FUNC_API 307
8.3.2.3 USB_VMSD_FILE_INFO 308
8.3.2.4 USB_VMSD_DIR_ENTRY_SHORT 309
8.3.3 Function definitions 311
8.3.3.1 USB_VMSD_ON_READ_FUNC 311
8.3.3.2 USB_VMSD_ON_WRITE_FUNC 312
8.3.3.3 USB_VMSD_MEM_ALLOC 313
8.3.3.4 USB_VMSD_MEM_FREE 314

9 Media Transfer Protocol Class (MTP) 315

9.1 Overview 316
9.1.1 Getting access to files 317
9.1.2 Additional information 320
9.1.3 Error handling on the host side 320
9.2 Configuration 321
9.2.1 Initial configuration 321
9.2.2 Final configuration 321
9.2.3 emFile and MTP configuration for UTF8 characters 321
9.2.4 Class specific configuration 321
9.2.5 Compile time configuration 321
9.3 Running the sample application 323
9.4 Target API 324
9.4.1 API functions 326
9.4.1.1 USBD_MTP_Add() 326
9.4.1.2 USBD_MTP_AddStorage() 327
9.4.1.3 USBD_MTP_RemoveStorage() 328
9.4.1.4 USBD_MTP_Task() 329
9.4.1.5 USBD_MTP_Poll() 330
9.4.1.6 USBD_MTP_SendEvent() 331
9.4.1.7 USBD_MTP_SetObjectAllocFailCb() 333
9.4.1.8 USBD_MTP_SetOperationCb() 334
9.4.1.9 USBD_MTP_SetWriteProtectionCb() 335

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.2 Data structures 336
9.4.2.1 USB_MTP_FILE_INFO 336
9.4.2.2 USB_MTP_INIT_DATA 337
9.4.2.3 USB_MTP_INFO 338
9.4.2.4 USB_MTP_INST_DATA 339
9.4.2.5 USB_MTP_INST_DATA_DRIVER 340
9.4.2.6 USB_MTP_STORAGE_API 341
9.4.2.7 USB_MTP_STORAGE_INFO 343
9.4.2.8 USB_MTP_OPERATION_INFO 344
9.4.3 Defines 345
9.4.3.1 Write protection attributes 345
9.4.4 Enums 346
9.4.4.1 USB_MTP_EVENT 346
9.4.4.2 USB_MTP_OPERATION_CB_TYPE 348
9.4.4.3 USB_MTP_STORAGE_CB_TYPE 349
9.4.5 Prototypes 350
9.4.5.1 USB_MTP_OBJECT_ALLOC_FAIL 350
9.4.5.2 USB_MTP_OPERATION_CB 351
9.4.5.3 USB_MTP_WP_CB 352
9.5 MTP Storage Driver 353
9.5.1 General information 353
9.5.2 Interface function list 353
9.5.3 USB_MTP_STORAGE_API in detail 354
9.5.3.1 USB_MTP_STORAGE_INIT 354
9.5.3.2 USB_MTP_STORAGE_GET_INFO 355
9.5.3.3 USB_MTP_STORAGE_FIND_FIRST_FILE 356
9.5.3.4 USB_MTP_STORAGE_FIND_NEXT_FILE 357
9.5.3.5 USB_MTP_STORAGE_OPEN_FILE 358
9.5.3.6 USB_MTP_STORAGE_CREATE_FILE 359
9.5.3.7 USB_MTP_STORAGE_READ_FROM_FILE 360
9.5.3.8 USB_MTP_STORAGE_WRITE_TO_FILE 361
9.5.3.9 USB_MTP_STORAGE_CLOSE_FILE 362
9.5.3.10 USB_MTP_STORAGE_REMOVE_FILE 363
9.5.3.11 USB_MTP_STORAGE_CREATE_DIR 364
9.5.3.12 USB_MTP_STORAGE_REMOVE_DIR 365
9.5.3.13 USB_MTP_STORAGE_FORMAT 366
9.5.3.14 USB_MTP_STORAGE_RENAME_FILE 367
9.5.3.15 USB_MTP_STORAGE_DEINIT 368
9.5.3.16 USB_MTP_STORAGE_GET_FILE_ATTRIBUTES 369
9.5.3.17 USB_MTP_STORAGE_MODIFY_FILE_ATTRIBUTES 370
9.5.3.18 USB_MTP_STORAGE_GET_FILE_CREATION_TIME 371
9.5.3.19 USB_MTP_STORAGE_GET_FILELAST_WRITE_TIME 372
9.5.3.20 USB_MTP_STORAGE_GET_FILE_ID 373
9.5.3.21 USB_MTP_STORAGE_GET_FILE_SIZE 374
9.5.3.22 USB_MTP_STORAGE_GET_FILE_INFO 375

10 Communication Device Class (CDC) 376

10.1 Overview 377
10.1.1 Configuration 377
10.1.2 CDC-ACM issues on Windows 10 377
10.2 The example application 379
10.2.1 Testing communication to the USB device 379
10.3 Target API 382
10.3.1 Interface function list 382
10.3.1.1 USBD_CDC_Add() 384
10.3.1.2 USBD_CDC_CancelRead() 385
10.3.1.3 USBD_CDC_CancelWrite() 386
10.3.1.4 USBD_CDC_Read() 387
10.3.1.5 USBD_CDC_ReadOverlapped() 388

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.6 USBD_CDC_Receive() 389
10.3.1.7 USBD_CDC_ReceivePoll() 390
10.3.1.8 USBD_CDC_ReadAsync() 391
10.3.1.9 USBD_CDC_SetOnBreak() 392
10.3.1.10 USBD_CDC_SetOnLineCoding() 393
10.3.1.11 USBD_CDC_SetOnControlLineState() 394
10.3.1.12 USBD_CDC_SetOnRXEvent() 395
10.3.1.13 USBD_CDC_SetOnTXEvent() 397
10.3.1.14 USBD_CDC_UpdateSerialState() 399
10.3.1.15 USBD_CDC_Write() 400
10.3.1.16 USBD_CDC_WriteAsync() 401
10.3.1.17 USBD_CDC_WaitForRX() 402
10.3.1.18 USBD_CDC_WaitForTX() 403
10.3.1.19 USBD_CDC_PollForTX() 404
10.3.1.20 USBD_CDC_WaitForTXReady() 405
10.3.1.21 USBD_CDC_PollForRX() 406
10.3.1.22 USBD_CDC_WaitForTX() 407
10.3.1.23 USBD_CDC_PollForTX() 408
10.3.1.24 USBD_CDC_WaitForTXReady() 409
10.3.1.25 USBD_CDC_WriteSerialState() 410
10.3.1.26 USBD_CDC_GetNumBytesRemToRead() 411
10.3.1.27 USBD_CDC_GetNumBytesRemToWrite() 412
10.3.1.28 USBD_CDC_GetNumBytesInBuffer() 413
10.3.2 Data structures 414
10.3.2.1 USB_CDC_INIT_DATA 414
10.3.2.2 USB_CDC_LINE_CODING 415
10.3.2.3 USB_CDC_SERIAL_STATE 416
10.3.2.4 USB_CDC_CONTROL_LINE_STATE 417

11 Human Interface Device Class (HID) 418

11.1 Overview 419
11.1.1 Further reading 419
11.1.2 Categories 419
11.1.2.1 True HIDs 419
11.1.2.2 Vendor specific HIDs 419
11.2 Background information 421
11.2.1 HID descriptors 421
11.2.1.1 HID descriptor 421
11.2.1.2 Report descriptor 421
11.2.1.3 Physical descriptor 422
11.3 Configuration 423
11.3.1 Initial configuration 423
11.3.2 Final configuration 423
11.4 Example application 424
11.4.1 USB_HID_Mouse.c 424
11.4.2 USB_HID_Echo1.c 424
11.4.2.1 Running the example 425
11.4.2.2 Compiling the PC example application 425
11.5 Target API 427
11.5.1 Target interface function list 428
11.5.2 HID Target API functions 429
11.5.2.1 USBD_HID_AddEx() 429
11.5.2.2 USBD_HID_Add() 430
11.5.2.3 USBD_HID_GetNumBytesInBuffer() 431
11.5.2.4 USBD_HID_GetNumBytesRemToRead() 432
11.5.2.5 USBD_HID_GetNumBytesRemToWrite() 433
11.5.2.6 USBD_HID_Read() 434
11.5.2.7 USBD_HID_ReadOverlapped() 435
11.5.2.8 USBD_HID_Receive() 436

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.9 USBD_HID_ReceivePoll() 437
11.5.2.10 USBD_HID_WaitForRX() 438
11.5.2.11 USBD_HID_WaitForTX() 439
11.5.2.12 USBD_HID_Write() 440
11.5.2.13 USBD_HID_SetOnGetReportRequest() 441
11.5.2.14 USBD_HID_SetOnSetReportRequest() 442
11.5.2.15 USBD_HID_ReadReport() 443
11.5.3 Data structures 444
11.5.3.1 USB_HID_INIT_DATA_EX 444
11.5.3.2 USB_HID_INIT_DATA 446
11.5.4 Type definitions 447
11.5.4.1 USB_HID_ON_GETREPORT_REQUEST_FUNC 447
11.5.4.2 USB_HID_ON_SETREPORT_REQUEST_FUNC 448
11.6 Host API 449
11.6.1 Host API function list 450
11.6.2 HID Host API functions 451
11.6.2.1 USBHID_Close() 451
11.6.2.2 USBHID_Open() 452
11.6.2.3 USBHID_Init() 453
11.6.2.4 USBHID_Exit() 454
11.6.2.5 USBHID_Read() 455
11.6.2.6 USBHID_Write() 456
11.6.2.7 USBHID_GetNumAvailableDevices() 457
11.6.2.8 USBHID_GetProductName() 458
11.6.2.9 USBHID_GetInputReportSize() 459
11.6.2.10 USBHID_GetOutputReportSize() 460
11.6.2.11 USBHID_GetProductId() 461
11.6.2.12 USBHID_GetVendorId() 462
11.6.2.13 USBHID_RefreshList() 463
11.6.2.14 USBHID_SetVendorPage() 464

12 Printer Class 465

12.1 Overview 466
12.1.1 Configuration 466
12.2 The example application 467
12.3 Target API 469
12.3.1 Interface function list 469
12.3.2 API functions 470
12.3.2.1 USB_PRINTER_Init() 470
12.3.2.2 USB_PRINTER_Task() 471
12.3.2.3 USB_PRINTER_TaskEx() 472
12.3.2.4 USB_PRINTER_ConfigIRQProcessing() 473
12.3.2.5 USB_PRINTER_Read() 474
12.3.2.6 USB_PRINTER_ReadTimed() 475
12.3.2.7 USB_PRINTER_Receive() 476
12.3.2.8 USB_PRINTER_ReceiveTimed() 477
12.3.2.9 USB_PRINTER_Write() 478
12.3.2.10 USB_PRINTER_WriteTimed() 479
12.3.2.11 USB_PRINTER_SetOnVendorRequest() 480
12.3.2.12 USB_PRINTER_SetClass() 481
12.3.2.13 USB_PRINTER_API 482
12.4 Printer API 483
12.4.1 General information 483
12.4.2 USB_PRINTER_API in detail 484
12.4.2.1 USB_PRINTER_GET_DEVICE_ID_STRING 484
12.4.2.2 USB_PRINTER_ON_DATA_RECEIVED 485
12.4.2.3 USB_PRINTER_GET_HAS_NO_ERROR 486
12.4.2.4 USB_PRINTER_GET_IS_SELECTED 487
12.4.2.5 USB_PRINTER_GET_IS_PAPER_EMPTY 488

 © 2010-2026 SEGGER Microcontroller GmbH

12.4.2.6 USB_PRINTER_ON_RESET 489

13 IP-over-USB (IP) 490

13.1 Overview 491
13.2 Using only RNDIS or CDC-ECM 492
13.2.1 Working with emUSB-Device-IP 492
13.3 Configuration 494
13.3.1 Initial Configuration 494
13.3.2 Final configuration 494
13.3.3 Class-specific configuration 494
13.4 Running the sample application 495
13.5 emUSB-Device-IP + emNet as a "USB Webserver" 496
13.6 Target API 497
13.6.1 API functions 498
13.6.1.1 USBD_IP_Add() 498
13.6.1.2 USBD_IP_Task() 499
13.6.2 Data structures 500
13.6.2.1 USB_IP_INIT_DATA 500

14 Remote NDIS (RNDIS) 501

14.1 Overview 502
14.1.1 Working with RNDIS 502
14.1.2 Additional information 502
14.2 Configuration 503
14.2.1 Initial Configuration 503
14.2.2 Final configuration 503
14.2.3 Class-specific configuration 503
14.3 Running the sample application 504
14.3.1 IP_Config_RNDIS.c in detail 504
14.4 RNDIS + emNet as a "USB Web Server" 506
14.5 Target API 507
14.5.1 API functions 508
14.5.1.1 USBD_RNDIS_Add() 508
14.5.1.2 USBD_RNDIS_Task() 509
14.5.1.3 USBD_RNDIS_SetDeviceInfo() 510
14.5.2 Data structures 511
14.5.2.1 USB_RNDIS_INIT_DATA 511
14.5.2.2 USB_RNDIS_DEVICE_INFO 512
14.5.3 Driver interface 513
14.5.3.1 USB_IP_NI_DRIVER_API 513
14.5.3.2 USB_IP_NI_DRIVER_DATA 514
14.6 RNDIS IP Driver 515
14.6.1 General information 515
14.6.2 Interface function list 515
14.6.3 USB_IP_NI_DRIVER_API in detail 516
14.6.3.1 USB_IP_NI_INIT 516
14.6.3.2 USB_IP_NI_GET_PACKET_BUFFER 517
14.6.3.3 USB_IP_NI_WRITE_PACKET 518
14.6.3.4 USB_IP_NI_SET_PACKET_FILTER 519
14.6.3.5 USB_IP_NI_GET_LINK_STATUS 520
14.6.3.6 USB_IP_NI_GET_LINK_SPEED 521
14.6.3.7 USB_IP_NI_GET_HWADDR 522
14.6.3.8 USB_IP_NI_GET_STATS 523
14.6.3.9 USB_IP_NI_GET_MTU 524
14.6.3.10 USB_IP_NI_RESET 525
14.6.3.11 USB_IP_NI_SET_WRITE_PACKET_FUNC 526
14.6.3.12 USB_IP_NI_SET_REPORT_LINKSTATE_FUNC 527

 © 2010-2026 SEGGER Microcontroller GmbH

15 CDC-ECM 528
15.1 Overview 529
15.1.1 Working with CDC-ECM 529
15.1.2 Additional information 530
15.2 Configuration 531
15.2.1 Initial configuration 531
15.2.2 Final configuration 531
15.3 Running the sample application 532
15.3.1 IP_Config_ECM.c in detail 532
15.4 Target API 534
15.4.1 API functions 535
15.4.1.1 USBD_ECM_Add() 535
15.4.1.2 USBD_ECM_Task() 536
15.4.2 Data structures 537
15.4.2.1 USB_ECM_INIT_DATA 537
15.4.3 Driver interface 538
15.4.3.1 USB_IP_NI_DRIVER_API 538
15.4.3.2 USB_IP_NI_DRIVER_DATA 539
15.5 CDC-ECM IP driver 540
15.5.1 General information 540
15.5.2 Interface function list 540
15.5.3 USB_IP_NI_DRIVER_API in detail 541
15.5.3.1 USB_IP_NI_INIT 541
15.5.3.2 USB_IP_NI_GET_PACKET_BUFFER 542
15.5.3.3 USB_IP_NI_WRITE_PACKET 543
15.5.3.4 USB_IP_NI_SET_PACKET_FILTER 544
15.5.3.5 USB_IP_NI_GET_LINK_STATUS 545
15.5.3.6 USB_IP_NI_GET_LINK_SPEED 546
15.5.3.7 USB_IP_NI_GET_HWADDR 547
15.5.3.8 USB_IP_NI_GET_STATS 548
15.5.3.9 USB_IP_NI_GET_MTU 549
15.5.3.10 USB_IP_NI_RESET 550
15.5.3.11 USB_IP_NI_SET_WRITE_PACKET_FUNC 551
15.5.3.12 USB_IP_NI_SET_REPORT_LINKSTATE_FUNC 552

16 CDC-NCM 553
16.1 Overview 554
16.1.1 Working with CDC-NCM 554
16.1.2 Additional information 554
16.2 Configuration 555
16.2.1 Initial configuration 555
16.2.2 Final configuration 555
16.3 Running the sample application 556
16.3.1 IP_Config_NCM.c in detail 556
16.4 Target API 558
16.4.1 API functions 559
16.4.1.1 USBD_NCM_Add() 559
16.4.2 Data structures 560
16.4.2.1 USB_NCM_INIT_DATA 560
16.4.3 Driver interface 561
16.4.3.1 USB_IP_NI_DRIVER_API 561
16.4.3.2 USB_IP_NI_DRIVER_DATA 562
16.5 CDC-NCM IP Driver 563
16.5.1 General information 563
16.5.2 Interface function list 563
16.5.3 USB_IP_NI_DRIVER_API in detail 564
16.5.3.1 USB_IP_NI_INIT 564
16.5.3.2 USB_IP_NI_GET_PACKET_BUFFER 565
16.5.3.3 USB_IP_NI_WRITE_PACKET 566

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.4 USB_IP_NI_SET_PACKET_FILTER 567
16.5.3.5 USB_IP_NI_GET_LINK_STATUS 568
16.5.3.6 USB_IP_NI_GET_LINK_SPEED 569
16.5.3.7 USB_IP_NI_GET_HWADDR 570
16.5.3.8 USB_IP_NI_GET_STATS 571
16.5.3.9 USB_IP_NI_GET_MTU 572
16.5.3.10 USB_IP_NI_RESET 573
16.5.3.11 USB_IP_NI_SET_WRITE_PACKET_FUNC 574
16.5.3.12 USB_IP_NI_SET_REPORT_LINKSTATE_FUNC 575

17 Audio 576

17.1 Overview 577
17.2 Creation of an audio device application 578
17.2.1 Configuration requirements 579
17.2.2 Design of audio interfaces 579
17.2.3 Handling of audio control requests 580
17.2.4 Receiving audio data 580
17.2.4.1 Using explicit feedback 580
17.2.5 Sending audio data 580
17.2.5.1 Using explicit feedback 581
17.2.6 Physical controls 581
17.3 Syntax definition of the USB audio design file 582
17.3.1 Overall syntax of the design file 583
17.3.1.1 Compiler Macros 583
17.3.2 Control units description 584
17.3.2.1 Input Terminal 584
17.3.2.2 Output Terminal 585
17.3.2.3 Feature unit 585
17.3.2.4 Mixer unit 586
17.3.2.5 Selector unit 586
17.3.2.6 Clock source 587
17.3.2.7 Clock selector 587
17.3.2.8 Clock multiplier 587
17.3.3 Streaming interface description 589
17.3.3.1 AUDIO_STREAM section 589
17.3.3.2 ENDPOINT section 590
17.3.4 Stream units description 591
17.3.4.1 Format I section 591
17.3.4.2 Format II section 591
17.3.4.3 Format III section 591
17.4 Target API 593
17.4.1 API functions 595
17.4.1.1 USBD_AC_Add() 595
17.4.1.2 USBD_AC_GetCurrentAltSetting() 596
17.4.1.3 USBD_AC_GetStreamInfo() 597
17.4.1.4 USBD_AC_OpenRXStream() 598
17.4.1.5 USBD_AC_CloseRXStream() 599
17.4.1.6 USBD_AC_OpenTXStream() 600
17.4.1.7 USBD_AC_Send() 601
17.4.1.8 USBD_AC_CloseTXStream() 602
17.4.1.9 USBD_AC_SetFeedbackDataRate() 603
17.4.1.10 USBD_AC_GetFeedbackDataRate() 604
17.4.1.11 USBD_AC_SendInterruptMessage() 605
17.4.2 Data structures 606
17.4.2.1 USBD_AC_INIT_DATA 606
17.4.2.2 USBD_AC_STREAM_INTF_INFO 607
17.4.2.3 USBD_AC_RX_CTX 608
17.4.2.4 USBD_AC_RX_DATA 609
17.4.2.5 USBD_AC_TX_CTX 610

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.2.6 USBD_AC_CONTROL_INFO 611
17.4.2.7 USBD_AC_EVENT 612
17.4.3 Function definitions 613
17.4.3.1 USBD_AC_SET_ALT_INTERFACE 613
17.4.3.2 USBD_AC_CONTROL_GET_FUNC 614
17.4.3.3 USBD_AC_CONTROL_SET_FUNC 615
17.4.3.4 USBD_AC_RX_CALLBACK 616
17.4.3.5 USBD_AC_TX_CALLBACK 617

18 Legacy Audio 1.0 618

18.1 Overview 619
18.2 Introduction 620
18.3 Configuration 621
18.3.1 Initial configuration 621
18.3.2 Final configuration 621
18.3.3 Using the microphone interface 621
18.3.4 Using the speaker interface 622
18.4 Target API 623
18.4.1 API functions 624
18.4.1.1 USBD_AUDIO_Add() 624
18.4.1.2 USBD_AUDIO_Read_Task() 625
18.4.1.3 USBD_AUDIO_Write_Task() 626
18.4.1.4 USBD_AUDIO_Start_Play() 627
18.4.1.5 USBD_AUDIO_Stop_Play() 628
18.4.1.6 USBD_AUDIO_Start_Listen() 629
18.4.1.7 USBD_AUDIO_Stop_Listen() 630
18.4.1.8 USBD_AUDIO_Set_Timeouts() 631
18.4.2 Data structures 632
18.4.2.1 USBD_AUDIO_INIT_DATA 632
18.4.2.2 USBD_AUDIO_IF_CONF 633
18.4.2.3 USBD_AUDIO_FORMAT 635
18.4.2.4 USBD_AUDIO_UNITS 636
18.4.3 Function definitions 637
18.4.3.1 USBD_AUDIO_TX_FUNC 637
18.4.3.2 USBD_AUDIO_RX_FUNC 638
18.4.3.3 USBD_AUDIO_CONTROL_FUNC 640

19 USB video device class (UVC) 643

19.1 Overview 644
19.2 Configuration 645
19.2.1 Initial configuration 645
19.2.1.1 Uncompressed video format 645
19.2.2 Final configuration 645
19.3 Target API 646
19.3.1 API functions 647
19.3.1.1 USBD_UVC_Add() 647
19.3.1.2 USBD_UVC_Write() 648
19.3.1.3 USBD_UVC_WriteEx() 650
19.3.1.4 USBD_UVC_SetOnResolutionChange() 651
19.3.2 Data structures 652
19.3.2.1 USBD_UVC_INIT_DATA 652
19.3.2.2 USBD_UVC_BUFFER 653
19.3.2.3 USBD_UVC_DATA_BUFFER 654
19.3.2.4 USBD_UVC_RESOLUTION 655
19.3.3 Function prototypes 656
19.3.3.1 USB_UVC_ON_RESOLUTION_CHANGE 656

 © 2010-2026 SEGGER Microcontroller GmbH

20 Device Firmware Upgrade (DFU) 657
20.1 Overview 658
20.1.1 Using DFU on Windows 658
20.2 Configuration 659
20.2.1 Dual configuration mode 659
20.2.2 Single configuration 659
20.3 Target API 660
20.3.1 API functions 661
20.3.1.1 USBD_DFU_Add() 661
20.3.1.2 USBD_DFU_Add_RunTime() 662
20.3.1.3 USBD_DFU_AddAlternateInterface() 663
20.3.1.4 USBD_DFU_SetMSDescInfo() 664
20.3.1.5 USBD_DFU_SetPollTimeout() 665
20.3.1.6 USBD_DFU_Ack() 666
20.3.1.7 USBD_DFU_SetError() 667
20.3.1.8 USBD_DFU_ManifestComplt() 668
20.3.1.9 USBD_DFU_GetStatusReqCnt() 669
20.3.1.10 USBD_DFU_GetAlternateSetting() 670
20.3.2 Data structures 671
20.3.2.1 USB_DFU_INIT_DATA 671
20.3.3 Function prototypes 672
20.3.3.1 USBD_DFU_DETACH_REQUEST 672
20.3.3.2 USBD_DFU_DOWNLOAD 673
20.3.3.3 USBD_DFU_UPLOAD 674

21 Musical Instrument Digital Interface (MIDI) 675
21.1 Overview 676
21.2 Introduction 677
21.3 Configuration 679
21.3.1 Initial configuration 679
21.3.2 Final configuration 679
21.3.3 Testing MIDI on different operating systems 679
21.4 Target API 680
21.4.1 API functions 681
21.4.1.1 USBD_MIDI_Init() 681
21.4.1.2 USBD_MIDI_Add() 682
21.4.1.3 USBD_MIDI_ReceivePackets() 683
21.4.1.4 USBD_MIDI_GetNumPacketsInBuffer() 684
21.4.1.5 USBD_MIDI_ConvertPackets() 685
21.4.1.6 USBD_MIDI_WritePackets() 686
21.4.1.7 USBD_MIDI_WriteStream() 687
21.4.2 Data structures 688
21.4.2.1 USBD_MIDI_INIT_DATA 688
21.4.2.2 USBD_MIDI_JACK 689
21.4.2.3 USBD_MIDI_PACKET 690

22 Smart Card Device Class (CCID) 691
22.1 Overview 692
22.2 Target API 693
22.2.1 API functions 694
22.2.1.1 USBD_CCID_Init() 694
22.2.1.2 USBD_CCID_Add() 695
22.2.1.3 USBD_CCID_ReceiveCmd() 696
22.2.1.4 USBD_CCID_SendStatus() 697
22.2.1.5 USBD_CCID_SendDataBlock() 698
22.2.1.6 USBD_CCID_SendEscape() 699
22.2.1.7 USBD_CCID_SendParameters() 700
22.2.1.8 USBD_CCID_SendDataRateAndClockFrequency() 701

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1.9 USBD_CCID_NotifySlotState() 702
22.2.1.10 USBD_CCID_NotifyHwError() 703
22.2.2 Data structures 704
22.2.2.1 USB_CCID_INIT_DATA 704
22.2.2.2 USB_CCID_PROPERTIES 705
22.2.2.3 USB_CCID_CMD 707
22.2.2.4 USB_CCID_PROTOCOL_DATA_T0 709
22.2.2.5 USB_CCID_PROTOCOL_DATA_T1 710
22.2.3 Function prototypes 711
22.2.3.1 USBD_CCID_ABORT_CB 711

23 emUSB-Web add-on 712

23.1 Overview 713
23.2 Requirements 714
23.3 Configuration 715
23.3.1 Initial configuration 715
23.3.1.1 emUSB-Web diagram 715
23.3.2 emUSB-Web operation in detail 715
23.3.2.1 Device recognition 716
23.3.2.2 emUSB-Web protocol 716

24 Combining USB components (Multi-Interface) 717

24.1 Overview 718
24.1.1 Single interface device classes 719
24.1.2 Multiple interface device classes 719
24.1.3 IAD class 719
24.2 Configuration 721
24.3 How to combine 722
24.4 emUSB-Device component-specific modification 726
24.4.1 CDC component 726
24.4.1.1 Device side 726
24.4.1.2 Host side 726
24.5 Issues on Windows 7 728
24.5.1 Detailed description 728

25 Target OS Interface 729

25.1 General information 730
25.1.1 Operating system support supplied with this release 730
25.2 Interface function list 731
25.2.1 USB_OS_DeInit() 732
25.2.2 USB_OS_Delay() 733
25.2.3 USB_OS_DecRI() 734
25.2.4 USB_OS_GetTickCnt() 735
25.2.5 USB_OS_IncDI() 736
25.2.6 USB_OS_Init() 737
25.2.7 USB_OS_Signal() 738
25.2.8 USB_OS_Wait() 739
25.2.9 USB_OS_WaitTimed() 740
25.2.10 USB_OS_MutexAlloc() 741
25.2.11 USB_OS_MutexFree() 742
25.2.12 USB_OS_MutexLock() 743
25.2.13 USB_OS_MutexUnlock() 744

26 Target USB Driver 745

26.1 General information 746
26.1.1 Available USB drivers 746
26.2 Adding a driver to emUSB-Device 747
26.2.1 USBD_X_Config() 747

 © 2010-2026 SEGGER Microcontroller GmbH

26.2.2 USBD_X_DisableInterrupt() 749
26.2.3 USBD_X_EnableInterrupt() 750
26.3 Device driver specifics 751
26.3.1 LPC54/55xxx full-speed driver 752
26.3.2 LPC54/55xxx high-speed driver 752
26.3.3 EHCI driver 753
26.3.4 Synopsys DWC2 driver (slave mode) 754
26.3.5 Synopsys DWC2 driver (DMA mode) 755
26.3.6 XHCI driver 756
26.3.7 Renesas RX driver 757
26.3.8 AT91RM9200 driver 758
26.3.9 Giga Device GD32F4xx driver (full-speed controller) 759
26.3.10 Giga Device GD32F4xx driver (high-speed controller) 759
26.3.11 Atmel ATSAMV7x driver 760
26.3.12 PSOC driver 761
26.3.12.1 Restrictions 761
26.3.12.2 PSOC driver specific functions 761
26.3.12.2.1 USB_DRIVER_Cypress_PSoC6_SysTick() 761
26.3.12.2.2 USB_DRIVER_Cypress_PSoC6_Resume() 761
26.3.12.2.3 USB_DRIVER_Cypress_PSoC6_ConfigDMA() 762
26.3.12.2.4 USB_DRIVER_Infineon_PSC3M6_SysTick() 762
26.3.12.2.5 USB_DRIVER_Infineon_PSC3M6_Resume() 762
26.3.12.2.6 USB_DRIVER_Infineon_PSC3M6_SetBaseAddr() 762
26.3.12.2.7 USB_DRIVER_Infineon_PSC3M6_ConfigDMA() 763
26.3.13 ST full-speed driver 764
26.3.14 PSOCE84 driver 765
26.3.14.1 PSOCE84 driver-specific functions 765
26.3.14.1.1 USB_DRIVER_Infineon_PSOCE84_DMA_SetPLLRefClock() 765
26.3.15 Atmel / Microchip high-speed driver 766

27 Support 767
27.1 Contacting support 768
27.1.1 Where can I find the license number? 768

28 Profiling with SystemView 769
28.1 Profiling overview 770
28.2 Additional files for profiling 771
28.2.1 Additional files on target side 771
28.2.2 Additional files on PC side 771
28.3 Enable profiling 772
28.4 Recording and analyzing profiling information 773

29 Debugging 774
29.1 Message output 775
29.2 API functions 776
29.2.1 USBD_AddLogFilter() 777
29.2.2 USBD_AddWarnFilter() 778
29.2.3 USBD_SetLogFilter() 779
29.2.4 USBD_SetWarnFilter() 780
29.2.5 USB_PANIC 781
29.2.6 USB_X_Log() 782
29.2.7 USB_X_Warn() 783
29.2.8 USB_OS_Panic() 784
29.3 Message types 785

30 Performance & resource usage 787
30.1 Memory footprint 788

 © 2010-2026 SEGGER Microcontroller GmbH

30.2 Performance 791
31 FAQ 792

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 1

Introduction

This chapter will give a short introduction to emUSB-Device, including the supported USB classes and components. Host and target requirements are covered as well.

 © 2010-2026 SEGGER Microcontroller GmbH

1.1 Overview

This guide describes how to install, configure and use emUSB-Device. It also explains the internal structure of emUSB-Device.

emUSB-Device has been designed to work on any embedded system with a USB client controller. It can be used with USB 1.1, USB 2.0 or USB 3.0 devices.

The highest possible transfer rate on USB 2.0 full-speed (12 Mbit/s) devices is approximately 1.2 MB/s. In USB 2.0 high-speed mode (480 MBit/s) transfer rates of approximately 42 MByte/s could be achieved. USB 3.0 SuperSpeed (5 Gbit/s) is also supported.

It depends on the capabilities of the USB controller hardware which USB version and actual speed can be used on an embedded system.

 © 2010-2026 SEGGER Microcontroller GmbH

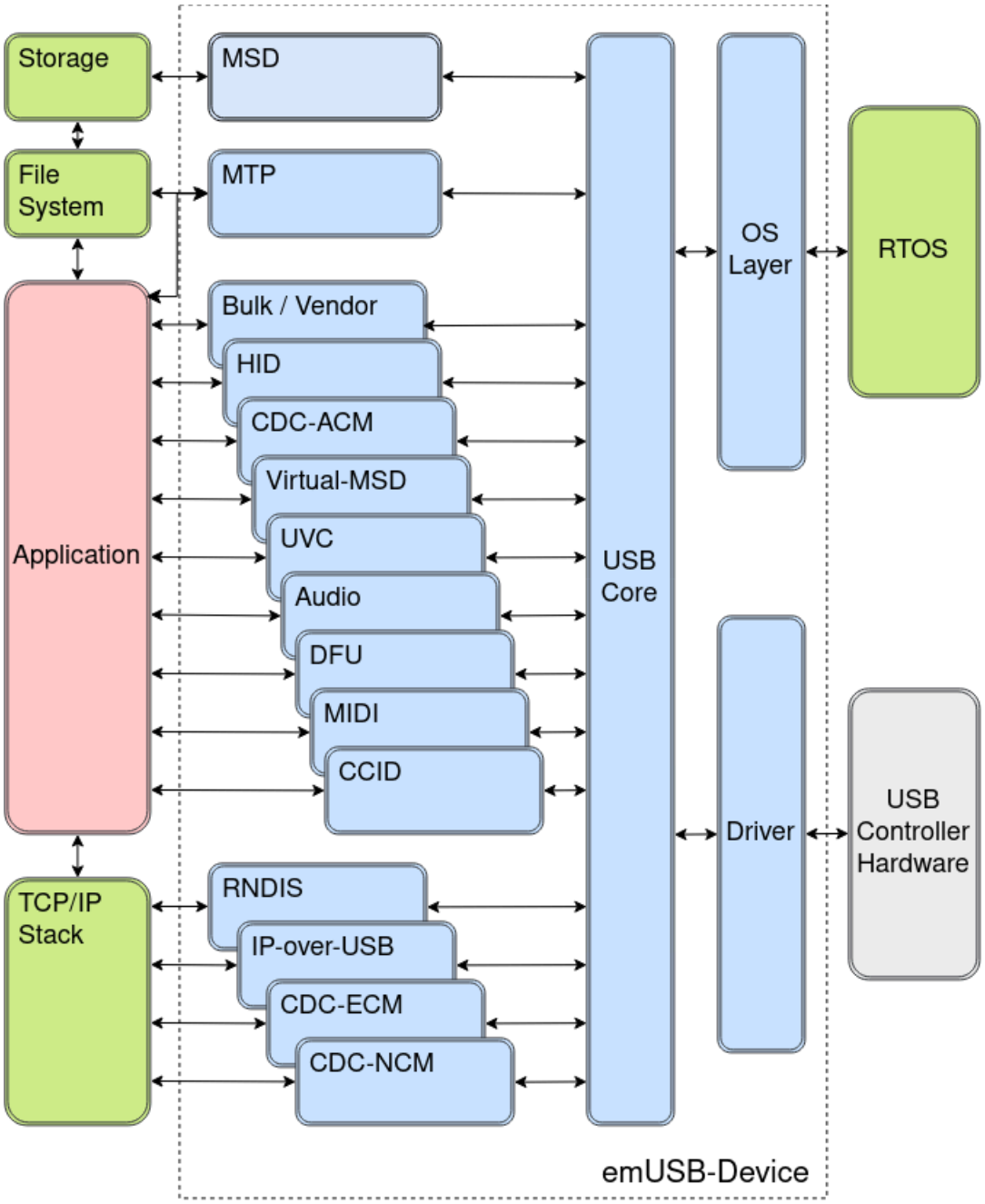
1.2 emUSB-Device features

Key features of emUSB-Device are:

- High performance
- Can be used with or without an RTOS
- Easy to use
- Easy to port
- No custom USB host driver necessary
- Start / test application supplied
- Highly efficient, portable, and commented ANSI C source code
- Hardware abstraction layer allows rapid addition of support for new devices

 © 2010-2026 SEGGER Microcontroller GmbH

1.3 System architecture



1.3.1 External components interfacing to emUSB-Device

1.3.1.1 Application

The application firmware on an embedded device that is using USB services.

1.3.1.2 File system

Middleware component that manages to store files in an organized manner (e.g. FAT) on non-volatile storage media. It usually needs a storage module to access the storage medium.

1.3.1.3 Storage module

Middleware component that stores data blocks on non-volatile storage media, like NAND flashes, SD-Cards, etc.

 © 2010-2026 SEGGER Microcontroller GmbH

1.3.1.4 TCP/IP stack

Middleware component implementing the TCP/IP protocol suite. It provides a socket interface for the application. The Ethernet driver interface of the TCP/IP stack is connected to a USB class for low-level packet transfer over USB.

1.3.1.5 RTOS

Real-time operating system providing synchronization services for the USB stack, especially events and mutexes. For all USB classes interfacing a TCP/IP stack, an RTOS is required. For all other USB classes, use of an RTOS is optional.

1.3.1.6 USB controller hardware

USB peripheral on the hardware implementing a physical USB bus.

 © 2010-2026 SEGGER Microcontroller GmbH

1.4 emUSB-Device components

emUSB-Device consists of three layers: A driver for hardware access, the emUSB-Device core and at least one USB class driver or the bulk communication component.

The different available hardware drivers, the USB class drivers, and the bulk communication component are additional packages, which can be combined and ordered as they fit to the requirements of your project. Normally, emUSB-Device consists of a driver that fits to the used hardware, the emUSB-Device core and at least one of the USB class drivers.

Component	Description
USB protocol layer	
Bulk / Vendor	emUSB-Device vendor component.
MSD	emUSB-Device Mass Storage Device class component.
IP-over-USB	emUSB-Device IP-over-USB component.
VirtualMSD	emUSB-Device VirtualMSD component.
CDC-ACM	emUSB-Device Communication Device Class component.
HID	emUSB-Device Human Interface Device Class component.
MTP	emUSB-Device Media Transfer Protocol component.
Printer	emUSB-Device Printer Class component.
RNDIS	emUSB-Device RNDIS component.
CDC-ECM	emUSB-Device CDC Ethernet Control Model component.
CDC-NCM	emUSB-Device CDC Network Control Model component.
UVC	emUSB-Device USB video class.
Audio	emUSB-Device USB audio class.
DFU	emUSB-Device Device Firmware Upgrade class.
MIDI	emUSB-Device Musical Instrument Digital Interface class.
CCID	emUSB-Device Smart Card Interface Device class.
Core layer	
emUSB-Device-Core	The emUSB-Device core is the intrinsic USB stack.
OS layer	Wrapper to interface a Real Time Operating system.
Hardware layer	
Driver	USB controller driver.

1.4.1 emUSB-Device-Bulk

emUSB-Device-Bulk allows you to quickly and smoothly develop software for an embedded device that communicates with a PC via USB. The communication is like a single, high-speed, reliable channel (very similar to a TCP connection). This bidirectional channel, with built-in flow control, allows the PC to send data to the embedded target, and the embedded target to receive these bytes and reply with any number of bytes. The PC is the USB host, and the target is the USB client.

1.4.2 emUSB-Device-MSD

1.4.2.1 Purpose of emUSB-Device-MSD

Access the target device like an ordinary disk drive

emUSB-Device-MSD enables the use of an embedded target device as a USB mass storage device. The target device can be simply plugged in and used like an ordinary disk drive, without the need to develop a driver for the host operating system. This is possible because

 © 2010-2026 SEGGER Microcontroller GmbH

the mass storage class is one of the standard device classes, defined by the USB Implementers Forum (USB IF). Virtually every major operating system on the market supports these device classes out of the box.

No custom host drivers necessary

Every major OS already provides host drivers for USB mass storage devices; there is no need to implement your own. The target device will be recognized as a mass storage device and can be accessed directly.

Plug and Play

Assuming the target system is a digital camera using emUSB-Device-MSD, videos or photos taken by this camera can be conveniently accessed with the file system explorer of the operating system in use when the camera is connected to the computer.

1.4.2.2 Typical applications

Typical applications are:

- Digital camera
- USB stick
- MP3 player
- DVD player

Any target with a USB interface: easy access to configuration and data files

1.4.2.3 emUSB-Device-MSD features

Key features of emUSB-Device-MSD are:

- Can be used with RAM, parallel flash, serial flash or mechanical drives
- Support for full-speed (12 Mbit/s) and high-speed (480 Mbit/s) transfer rates
- OS-abstraction: Can be used with any RTOS, but no OS is required for MSD-only devices

1.4.2.4 How does it work?

Use file system support from host OS

A device which uses emUSB-Device-MSD will be recognized as a mass storage device and can be used like an ordinary disk drive. If the device is unformatted when plugged in, the host operating system will ask you to format the device. Any file system provided by the host can be used. Typically, FAT is used, but other file systems such as NTFS are possible, too. If one of those file systems is used, the host is able to read from and write to the device using the storage functions of the emUSB-Device MSD component, which define unstructured read and write operations. Thus, there is no need to develop extra file system code if the application only accesses data on the target from the host side. This is typically the case for simple storage applications, such as USB memory sticks or ATA to USB bridges.

Provide file system code on the target if necessary

There are basically two types of MSD devices, one is where the device does not need to access the storage (e.g., USB stick, external HDD). The other type is where the device needs to write data onto the storage medium before it is accessed from a PC (e.g., data logger) or read data from it after it has been written onto the storage medium by a PC (e.g., an mp3 player or a device which reads configuration files from the storage). If you are using emUSB-Device-MSD you are most likely writing software for the former device type. emUSB-Device-MSD does not offer file-level access to the storage medium. You need a file system to access the storage. Developing a file system is a complex and time-consuming task and increases the time-to-market. Thus, we recommend the use of a commercial file system like *emFile*, SEGGER's file system for embedded applications. *emFile* is a high-performance library that is optimized for minimum memory consumption in RAM and ROM, high-speed and versatility. It is written in ANSI C and runs on any CPU and on any media. Refer to https://www.segger.com/emfile for more information about *emFile*.

 © 2010-2026 SEGGER Microcontroller GmbH

1.4.3 emUSB-Device IP-over-USB

emUSB-Device IP-over-USB allows running any IP-based protocol over USB. This component combines the advantages of RNDIS and CDC-ECM and allows plug-and-play on any major host operating system. Using the IP-over-USB technology in combination with a built-in web server, the device can easily be accessed from any host (Windows, Linux, Mac) by simply typing the device name into the web browser.

1.4.3.1 Typical applications

Typical applications are:

- Headphones
- Printer
- Data logger
- Ethernet2USB adapter

1.4.4 emUSB-Device-VirtualMSD

The emUSB-Device-VirtualMSD component allows you to easily stream files to and from USB devices. Once the USB device is connected to the host, files can be read or written to the application without the need for dedicated storage memory.

1.4.4.1 Typical applications

Typical applications are:

- Updating firmware (e.g. Handheld Terminal)
- Updating configuration files

1.4.5 emUSB-Device-CDC

emUSB-Device-CDC converts the target device into a serial communication device. A target device running emUSB-Device-CDC is recognized by the host as a serial interface (USB2COM, virtual COM port), without the need to install a special host driver, because the communication device class is one of the standard device classes and every major operating system already provides host drivers for those device classes. All PC software using a COM port will work without modifications with this virtual COM port.

1.4.5.1 Typical applications

Typical applications are:

- Modem
- Telephone system
- Fax machine

1.4.6 emUSB-Device-HID

The Human Interface Device class (HID) is an abstract USB class protocol defined by the USB Implementers Forum. This protocol was defined for handling devices that humans use to control the operation of computer systems. An installation of a custom host USB driver is not necessary because the USB Human Interface Device class is standardized, and every major OS already provides host drivers for it.

1.4.6.1 Typical applications

Typical applications are:

- Keyboard
- Mouse and similar pointing devices
- Gamepad
- Front-panel controls - for example, switches and buttons

 © 2010-2026 SEGGER Microcontroller GmbH

- Barcode reader
- Thermometer
- Voltmeter
- Low-speed JTAG emulator
- Uninterruptible power supply (UPS)

1.4.7 emUSB-Device-MTP

The Media Transfer Protocol (MTP) is a USB class protocol which can be used to transfer files to and from storage devices. MTP is an alternative to MSD as it operates on a file level rather than on a storage sector level. The advantage of MTP is the ability to access the storage medium from the host PC and from the device at the same time. Because MTP works at the file level, this also eliminates the risk of damaging the file system when the communication to the host has been canceled unexpectedly (e.g., the cable was removed). MTP is supported by most operating systems without the need to install third-party drivers.

1.4.7.1 Typical applications

Typical applications are:

- Digital camera
- USB stick
- MP3 player
- DVD player
- Telephone

Any target with a USB interface: easy access to configuration and data files.

1.4.8 emUSB-Device-Printer

emUSB-Device-Printer converts the target device into a printing device. A target device running emUSB-Device-Printer is recognized by the host as a printer. Unless the device identifies itself as a printer already recognized by the host PC, you must install a driver to be able to communicate with the USB device.

1.4.8.1 Typical applications

Typical applications are:

- Laser/Inkjet printer
- CNC machine

1.4.9 emUSB-Device-RNDIS

emUSB-Device-RNDIS allows you to create a virtual Ethernet adapter through which the host PC can communicate with the device using the Internet protocol suite (TCP, UDP, FTP, HTTP, Telnet). This allows the creation of USB-based devices which can host a web server or act as a telnet terminal or an FTP server. emUSB-Device-RNDIS offers a unique customer experience and allows you to save development and hardware cost by e.g., using a website as a user interface instead of creating an application for every major OS and by eliminating the Ethernet hardware components from your device.

1.4.9.1 Typical applications

Typical applications are:

- USB-Webserver
- USB-Terminal (e.g. Telnet)
- USB-FTP-Server

 © 2010-2026 SEGGER Microcontroller GmbH

1.4.10 emUSB-Device-CDC-ECM

emUSB-Device-CDC-ECM enables the creation of a virtual Ethernet adapter through which the host PC can communicate with the device using the Internet protocol suite (TCP, UDP, FTP, HTTP, Telnet). This allows the creation of USB-based devices which can host a web server or act as a telnet terminal or an FTP server. emUSB-Device-CDC-ECM offers a unique customer experience and enables saving development and hardware costs by e.g., using a website as a user interface instead of creating an application for every major OS and by eliminating the Ethernet hardware components from your device.

1.4.10.1 Typical applications

Typical applications are:

- USB-Webserver
- USB-Terminal (e.g., Telnet)
- USB-FTP-Server

1.4.11 OS Layer

The OS layer is a wrapper to interface an RTOS (real-time operating system). It is used by the USB stack to request synchronization services from the RTOS like events and mutexes. Segger provides a couple of different OS layers for common embedded operating systems. For use in a bare metal system without an RTOS, there is also a “dummy” OS layer that implements “busy waiting” for synchronization.

 © 2010-2026 SEGGER Microcontroller GmbH

1.5 Requirements

1.5.1 Target system

Hardware

The target system must have a USB controller. The memory requirements can be found in the chapter *Performance & resource usage* on page 787. In order to control when the device is enumerated by the host, a switchable attach is necessary. This is a switchable pull-up connected to the D+ line of USB.

Software

emUSB-Device is optimized to be used with *embOS* but works with any other supported RTOS or without an RTOS in a superloop. For information regarding the OS integration, refer to the chapter *Target OS Interface* on page 729.

1.5.2 Development environment (compiler)

The CPU used is of no importance; only an ANSI-compliant C compiler complying with at least one of the following international standards is required:

- ISO/IEC 9899:1999 (C99)
- ISO/IEC 14882:1998 (C++)

A C++ compiler is not required, but can be used. The application program can therefore also be programmed in C++ if desired.

 © 2010-2026 SEGGER Microcontroller GmbH

1.6 File structure

The following table shows the contents of the emUSB-Device root directory:

Directory	Contents
`Application`	Contains the application programs. Depending on which stack is used, several files are available for each stack. Detailed information can be found in the corresponding chapter.
`BSP`	Contains example hardware-specific configurations for different eval boards.
`Config`	Contains configuration files (`USB_Conf.h`, `USB_ConfigIO.c`).
`Doc`	Contains the emUSB-Device documentation.
`Inc`	Contains include files.
`Sample`	Contains operating-system-dependent files which allow running emUSB-Device with different RTOS's.
`SEGGER`	Contains generic routines from SEGGER.
`USB`	Contains the emUSB-Device source code.
`Windows`	Contains host-specific applications (for Windows, Linux, macOS) which can be used in conjunction with the device application samples.

 © 2010-2026 SEGGER Microcontroller GmbH

1.7 Multithreading

The emUSB target API is not generally thread-safe. But it is allowed to handle different endpoints in different tasks in parallel. Examples are:

- A task that performs all reads of data from the host while another task sends data to the host.
- Operating on different interfaces (e.g. a BULK and a CDC interface) in independent tasks.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 2

Background information

This is a short introduction to USB. The fundamentals of USB are explained and links to additional resources are given.

Information provided in this chapter is not required to use the software.

 © 2010-2026 SEGGER Microcontroller GmbH

2.1 USB

2.1.1 Short Overview

The Universal Serial Bus (USB) is a bus architecture for connecting multiple peripherals to a host computer. It is an industry standard — maintained by the USB Implementers Forum — and because of its many advantages it enjoys a huge industry-wide acceptance. Over the years, a number of USB-capable peripherals appeared on the market, for example printers, keyboards, mice, digital cameras etc. Among the top benefits of USB are:

- Excellent plug-and-play capabilities allow devices to be added to the host system without reboots ("hot-plug"). Plugged-in devices are identified by the host and the appropriate drivers are loaded instantly.
- USB allows easy extensions of host systems without requiring host-internal extension cards.
- Device bandwidths may range from a few kB/s to hundreds of MB/s.
- A wide range of packet sizes and data transfer rates are supported.
- USB provides internal error handling. Together with the already-mentioned hot-plug capability this greatly improves robustness.
- The provisions for powering connected devices dispense with the need for extra power supplies for many low-power devices.
- Several transfer modes are supported, which ensures the wide applicability of USB.

These benefits did not only lead to broad market acceptance but also added several advantages, such as low costs of USB cables and connectors or a wide range of USB stack implementations. Last but not least, the major operating systems such as Microsoft Windows, Mac OS X, or Linux provide excellent USB support.

2.1.2 Important USB standard versions

USB 1.1 (September 1998)

This standard version supports isochronous and asynchronous data transfers. It has dual-speed data transfer of 1.5 Mbit/s for low-speed and 12 Mbit/s for full-speed devices. The maximum cable length between host and device is five meters. Up to 500 mA of electric current may be distributed to low-power devices.

USB 2.0 (April 2000)

As with all previous USB standards, USB 2.0 is fully forward and backward compatible. Existing cables and connectors may be reused. A new high-speed transfer speed of 480 Mbit/s (40 times faster than USB 1.1 at full-speed) was added.

USB 3.0 (November 2008)

As all previous USB standards, USB 3.0 is fully forward and backward compatible. Existing cables and connectors may be reused, but the new speed can only be used with new USB 3.0 cables and devices. The new speed class is named USB Super-Speed, which offers a maximum rate of 5 Gbit/s.

USB 3.1 (July 2013)

As all previous USB standards, USB 3.1 is fully forward and backward compatible. The new specification replaces the 3.0 standard and introduces new transfer speeds of up to 10 Gbit/s.

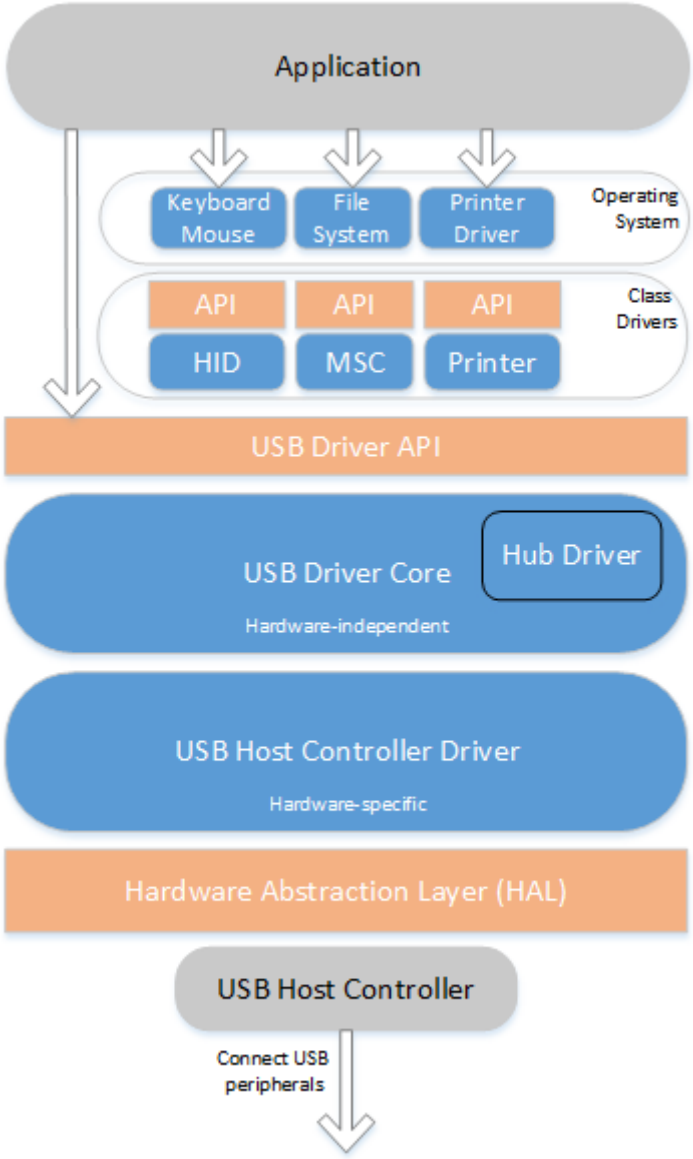
 © 2010-2026 SEGGER Microcontroller GmbH

2.1.3 USB system architecture

A USB system is composed of three parts - a host side, a device side and a physical bus. The physical bus is represented by the USB cable and connects the host and the device. The USB system architecture is asymmetric. Every single host can be connected to multiple devices in a tree-like fashion using special hub devices. You can connect up to 127 devices to a single host, but the count must include the hub devices as well.

A USB host consists of USB host controller hardware and a layered software stack. This host stack contains:

- A host controller driver (HCD) which provides the functionality of the host controller hardware.
- The USB Driver (USBD) Layer which implements the high-level functions used by USB device drivers in terms of the functionality provided by the HCD.
- The USB device drivers which establish connections to USB devices. The driver classes are also located here and provide generic access to certain types of devices such as printers or mass storage devices.



 © 2010-2026 SEGGER Microcontroller GmbH

USB Device

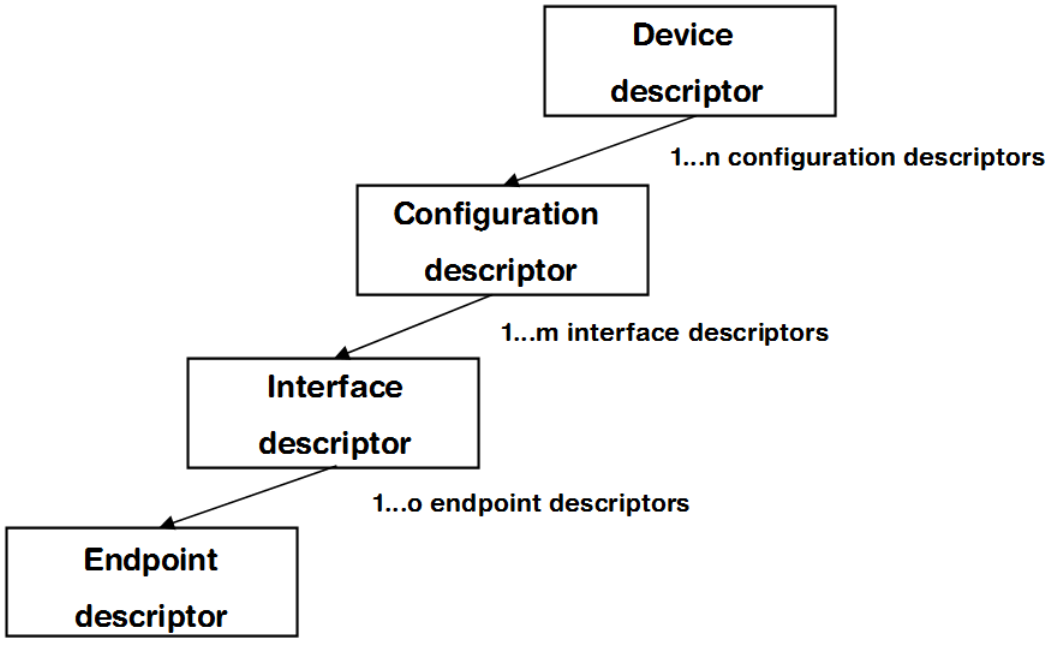
Two types of devices exist: hubs and functions. Hubs provide additional USB attachment points. Functions provide capabilities to the host and are able to transmit or receive data or control information over the USB bus. Every peripheral USB device represents at least one function but may implement more than one function. A USB printer for instance may provide file-system-like access in addition to printing.

In this guide we treat the term USB device as synonymous with functions and will not consider hubs.

Each USB device contains configuration information which describes its capabilities and resource requirements. A USB device must be configured by the host before its functions can be used. When a new device is connected for the first time, the host enumerates it, requests the configuration from the device, and performs the actual configuration. For example, if an embedded device uses emUSB-Device-MSD, the embedded device will appear as a USB mass storage device, and the host OS provides the driver out of the box. In general, there is no need to develop a custom driver to communicate with target devices that use one of the USB class protocols.

Descriptors

A device reports its attributes via descriptors. Descriptors are data structures with a standard-defined format. A USB device has one device descriptor which contains information applicable to the device and all of its configurations. It also contains the number of configurations the device supports. For each configuration, a configuration descriptor contains configuration-specific information. The configuration descriptor also contains the number of interfaces provided by the configuration. An interface groups the endpoints into logical units. Each interface descriptor contains information about the number of endpoints. Each endpoint has its own endpoint descriptor which states the endpoint's address, transfer types, etc.



As can be seen, the descriptors form a tree. The root is the device descriptor with n configuration descriptors as children, each of which has m interface descriptors which in turn have o endpoint descriptors each.

 © 2010-2026 SEGGER Microcontroller GmbH

2.1.4 Transfer types

The USB standard defines four transfer types: control, isochronous, interrupt, and bulk. Control transfers are used in the setup phase. The application can select one of the other three transfer types. For most embedded applications, bulk is the best choice because it allows the highest possible data rates.

Control transfers

Typically used for configuring a device when attached to the host. It may also be used for other device-specific purposes, including control of other pipes on the device.

Interrupt transfers

Typically used by devices that need guaranteed quick responses (fixed latency).

Bulk transfers

Typically used by devices that generate or consume data in relatively large and bursty quantities. Bulk transfer has wide dynamic latitude in transmission constraints. It can use all remaining available bandwidth, but with no guarantees on bandwidth or latency. Because the USB bus is normally not very busy, there is typically 90% or more of the bandwidth available for USB transfers.

Isochronous transfers

Typically used for applications which need guaranteed speed. Isochronous transfer offers a guaranteed bandwidth but with possible data loss. A typical use is for audio data which requires a constant data rate. Unlike bulk, control, or interrupt transfers, isochronous transfers do not receive an "ACK" from the other side; therefore, the sender does not know whether the data was received by the other side correctly. For applications where constant data rate is more important than data integrity (audio, video), the potential data loss does not pose an issue.

2.1.5 Setup phase / enumeration

The host first needs to get information from the target before the target can start communicating with the host. This information is gathered in the initial setup phase. The information is contained in the descriptors, which are in the configurable section of the USB-MSD stack. The most important parts of target device identification are the Product and Vendor IDs. During the setup phase, the host also assigns an address to the client. This part of the setup is called *enumeration*.

2.1.6 Product / Vendor IDs

The Product and Vendor IDs are necessary to identify the USB device. The Product ID describes a specific device type and does not need to be unique between different devices of the same type. USB host systems like Windows use the Product ID/Vendor ID combination to identify which drivers are needed.

For example: all our J-Link devices have the Vendor ID `0x1366` and Product ID `0x0105`.

A Vendor and Product ID is necessary only when development of the product is finished; during the development phase, the supplied Vendor and Product IDs can be used as samples. Using the sample Vendor ID (`0x8765`) or the SEGGER Vendor ID in a finished product is not allowed.

Possible options to obtain a Vendor ID or Product ID are described in the chapter *Vendor and Product ID* on page .

 © 2010-2026 SEGGER Microcontroller GmbH

2.2 Predefined device classes

The USB Implementers Forum has defined device classes for different purposes. In general, every device class defines a protocol for a particular type of application such as a mass storage device (MSD), human interface device (HID), etc. Device classes provide a standardized way of communication between host and device and typically work with a class driver which comes with the host operating system.

Using a predefined device class where applicable minimizes the amount of work to make a device usable on different host systems.

 © 2010-2026 SEGGER Microcontroller GmbH

2.3 USB hardware analyzers

A variety of USB hardware analyzers are on the market with different capabilities. If you are developing an application using emUSB-Device, it should not be necessary to have a USB analyzer, but we still recommend you do.

 © 2010-2026 SEGGER Microcontroller GmbH

2.4 References

For additional information, see the following documents:

- Universal Serial Bus Specification, Revision 2.0
- Universal Serial Bus Mass Storage Class Specification Overview, Rev 1.2
- UFI command specification: USB Mass Storage Class, UFI Command Specification, Rev 1.0

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 3

Getting started

The first step in getting emUSB-Device up and running is typically to compile it for the target system and to run it in the target system. This chapter explains how to do this.

 © 2010-2026 SEGGER Microcontroller GmbH

3.1 How to set up your target system

We assume that you are familiar with the tools you have selected for your project (compiler, project manager, linker, etc.). You should therefore be able to add files, add directories to the include search path, and so on. In this document the Embedded Studio IDE is used for all examples and screenshots, but every other ANSI C toolchain can also be used. It is also possible to use makefiles; in this case, when we say "add to the project", this translates into "add to the makefile".

Procedure to follow

Integration of emUSB-Device is a relatively simple process, which consists of the following steps:

- Take a running project for your target hardware.
- Add emUSB-Device files to the project.
- Add hardware-dependent configuration to the project.
- Prepare and run the application.

3.1.1 Take a running project

The project to start with should include the setup for basic hardware (e.g., CPU, PLL, DDR SDRAM) and initialization of the RTOS. emUSB-Device is designed to be used with embOS, SEGGER's real-time operating system. We recommend starting with an embOS sample project and including emUSB-Device in this project.

3.1.2 Add emUSB-Device files

Add all necessary source files from the `USB` folder to your project. You may simply add all files and let the linker drop everything not needed for your configuration. But there are some source files containing dependencies to emFile or emNet. If you don't have these middleware components, remove the respective files from your project.

Add RTOS layer

Additionally add the RTOS interface layer to your project. Choose a file from the folder `Sample/USB/OS` that matches your RTOS. For embOS use `USB_OS_embOSv5.c`. There is also a file `USB_OS_None.c` containing a layer to be used for superloop applications without an RTOS.

Configuring the include path

The include path is the path in which the compiler looks for include files. In cases where the included files (typically header files, .h) do not reside in the same folder as the C file to compile, an include path needs to be set. In order to build the project with all added files, you will need to add the following directories to your include path:

- `Config`
- `Inc`
- `SEGGER`
- `USB`

3.1.3 Configuring debugging output

While developing and testing emUSB-Device, we recommend using the DEBUG configuration of emUSB-Device. This is enabled by setting the preprocessor symbol `DEBUG` to 1 (or `USB_DEBUG_LEVEL` to 2). The DEBUG configuration contains many additional run-time checks and generates debug output messages which are very useful to identify problems that may occur during development. In case of a fatal problem (e.g., an invalid configuration), the program will end up in the function *`USB_OS_Panic()`* with an appropriate error message that describes the cause of the problem. Once the application is running correctly, `DEBUG` can be set to 0.

 © 2010-2026 SEGGER Microcontroller GmbH

Add the file USB_ConfigIO.c found in the folder Config to your project and configure it to match the message output method used by your debugging tools. If possible, use RTT.

To later compile a release configuration, which has a significantly smaller code footprint, simply set the preprocessor symbol DEBUG (or USB_DEBUG_LEVEL) to 0.

3.1.4 Add hardware dependent configuration

To perform target hardware dependent runtime configuration, the emUSB-Device stack calls a function named USBD_X_Config. Typical tasks that may be done inside this function are:

- Select an appropriate driver for the USB controller.
- Configure I/O pins of the MCU for USB.
- Configure PLL and clock divider necessary for USB operation.
- Install an interrupt service routine for USB.

Details can be found in *Target USB Driver* on page 745.

Sample configurations for popular evaluation boards are supplied with the driver shipment. They can be found in files called USB_Config_<TargetName>.c in the folders BSP/<Board-Name>/Setup.

Add the appropriate configuration file to your project. If there is no configuration file for your target hardware, take a file for similar hardware and modify it if necessary.

If the file needs modifications, we recommend copying it into the directory Config for easy updates to later versions of emUSB-Device.

Add BSP file

Some targets require CPU-specific functions for initialization, mainly for installing an interrupt service routine. They are contained in the file BSP_USB.c. Sample BSP_USB.c files for popular evaluation boards are supplied with the driver shipment. They can be found in the folders BSP/<BoardName>/Setup.

Add the appropriate BSP_USB.c file to your project. If there is no BSP file for your target hardware, take a file for similar hardware and modify it if necessary.

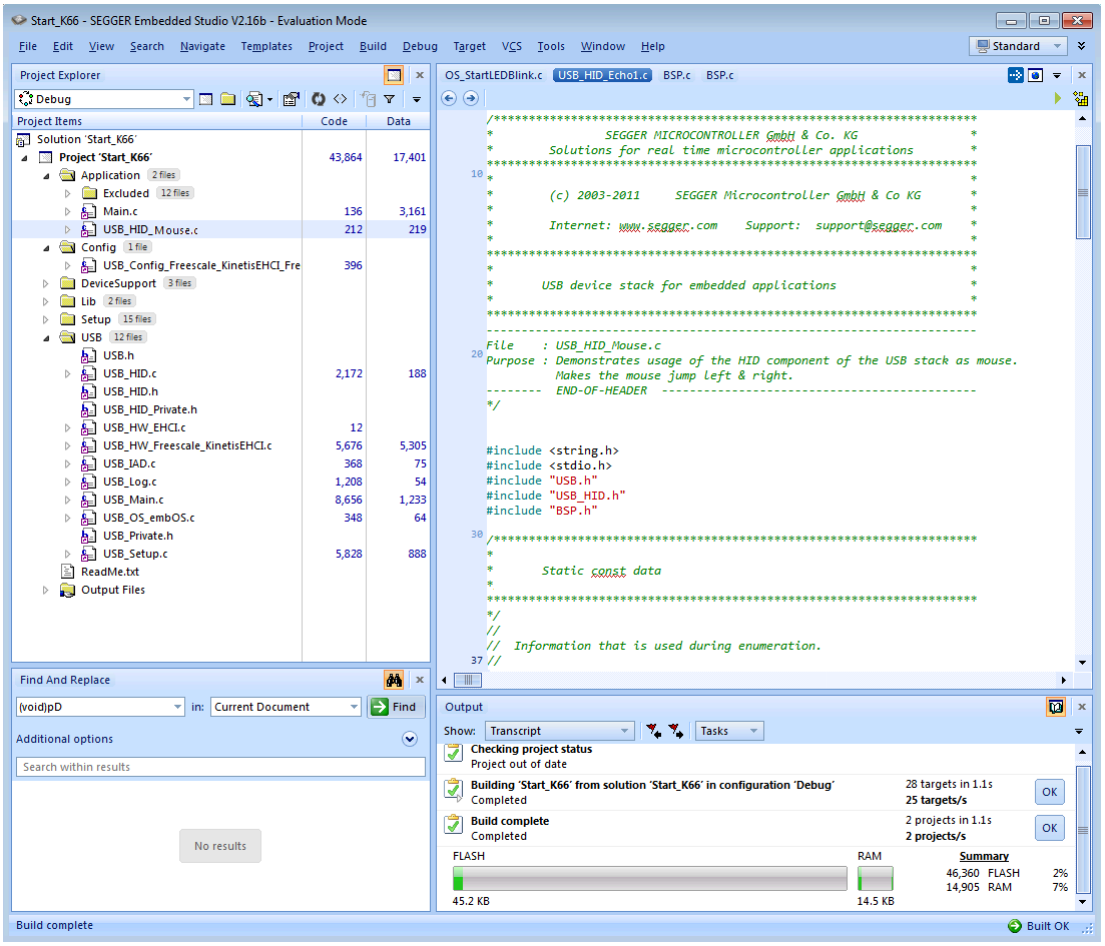
If the file needs modifications, we recommend copying it into the directory Config for easy updates to later versions of emUSB-Device.

Note that a BSP_USB.c file is not always required, because for some target hardware all runtime configuration is done in USB_X_Config.

3.1.5 Prepare and run the application

Choose a sample application from the folder Application and add it to your project. For example, add USB_HID_Mouse.c as your application to your project. Compile and run the application on the target hardware. After connecting the USB cable to the target device, the mouse pointer should hop from left to right.

 © 2010-2026 SEGGER Microcontroller GmbH



 © 2010-2026 SEGGER Microcontroller GmbH

3.2 Updating emUSB-Device

If an existing project should be updated to a later emUSB-Device version, only files have to be replaced. You should have received the emUSB-Device update as a zip file. Unzip this file to the location of your choice and replace all emUSB-Device files in your project with the newer files from the emUSB-Device update shipment.

In general, all files from the following directories have to be updated:

- `USB`
- `Inc`
- `SEGGER`
- `Doc`
- `Sample/USB/OS`

Some files may contain modifications required for project-specific customization. These files should reside in the folder `Config` and must not be overwritten. This includes:

- `USB_Conf.h`
- `USB_ConfigIO.c`
- `BSP_USB.c`
- `USB_Config_<TargetName>.c`

 © 2010-2026 SEGGER Microcontroller GmbH

3.3 emUSB-Device Configuration

An application using emUSB-Device must contain a `USB_DEVICE_INFO` structure containing the device identification information.

3.3.1 USB_DEVICE_INFO

Description

Device information that must be provided by the application via the function `USBD_SetDeviceInfo()` before the USB stack is started using `USBD_Start()`. Is used during enumeration of the device by the host.

Type definition

```
typedef struct {
  U16           VendorId;
  U16           ProductId;
  const char  * sVendorName;
  const char  * sProductName;
  const char  * sSerialNumber;
} USB_DEVICE_INFO;
```

Structure members

Member	Description
`VendorId`	Vendor ID. Uniquely identifies the vendor on a USB device.
`ProductId`	Product ID. Uniquely identifies all USB devices of a vendor.
`sVendorName`	Vendor name. ASCII string of up to 126 characters.
`sProductName`	Description of the USB device. ASCII string of up to 126 characters.
`sSerialNumber`	Serial number of the USB device (ASCII string). May be `NULL` if no serial number should be provided.

Additional information

The Product ID in combination with the Vendor ID creates a worldwide unique identifier for the product model. The Vendor ID is assigned by the USB Implementers Forum (https://www.usb.org). For tests, the default number above (or pretty much any other number) can be used. However, you may not bring a product to market without having been assigned your own Vendor ID. For emUSB-Device-CDC: If you change this value, do not forget to make the same change to the `.inf` file as described in section *The .inf file* on page . Otherwise, the Windows host will be unable to locate the driver.

The manufacturer name, product name and serial number are used during the enumeration phase. They together should give detailed information about which device is connected to the host.

> **Note**
>
> The max string length cannot be more than 126 ANSI characters.

Note for MSD: In order to conform to the USB bootability specification, the minimum string length of the serial number must be 12 characters where each character is a hexadecimal digit (’0’ through ’9’ or ’A’ through ’F’).

Example

```
static const USB_DEVICE_INFO _DeviceInfo = {
  0x8765,         // VendorId
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
  0x1234,          // ProductId
  "Vendor",        // VendorName
  "Bulk device",   // ProductName
  "13245678"       // SerialNumber
}
...
USBD_SetDeviceInfo(&_DeviceInfo);
...
USBD_Start();
```

This structure and functions are included in every example application and can be used without modifications in the development phase of your application, but you may not bring a product on the market without modifying the Vendor ID and Product ID.

Ids	Description
Default Vendor ID for all applications	
0x8765	Example Vendor ID for all examples. Do not use this in real products!
Used Product IDs	
0x1240	Example Product ID for all bulk samples.
0x1234	Example Product ID for deprecated bulk samples (using SEGGER Windows driver)
0x1200	Example Product ID for the MSD CD-ROM sample.
0x1000	Example Product ID for all MSD samples.
0x1088	Example Product ID for all UVC samples.
0x1111	Example Product ID for all CDC samples.
0x1112	Example Product ID for HID mouse sample.
0x1114	Example Product ID for the vendor specific HID sample.
0x1115	Example Product ID for HID keyboard sample.
0x1310	Example Product ID for the Audio Speaker sample.
0x1311	Example Product ID for the Audio Microphone sample.
0x1312	Example Product ID for the Audio Headset sample.
0x1350	Example Product ID for the MIDI sample.
0x2114	Example Product ID for the Printer class sample.
0x3000	Example Product ID for RNDIS sample.
0x3003	Example Product ID for ECM sample.
0x3004	Example Product ID for IP-over-USB sample.
0x3005	Example Product ID for NCM sample.

3.3.2 Additional required configuration for emUSB-MSD

Refer to *MSD Configuration* on page 252 for more information about the required additional configuration functions for emUSB-MSD.

3.3.3 Descriptors

All configuration descriptors are automatically generated by emUSB-Device and do not require configuration.

Some optional descriptors may be enabled by calling the following functions:

- `USBD_EnableIAD()`
- `USBD_UseV210()`

 © 2010-2026 SEGGER Microcontroller GmbH

- `USBD_EnableSuperSpeed()`
- `USBD_SetWebUSBInfo()`

 © 2010-2026 SEGGER Microcontroller GmbH

3.4 Compile-time configuration

emUSB-Device can be used without changing any of the compile-time switches. All compile-time configuration switches are preconfigured with valid values which match the requirements of most applications. An exception is the audio and video classes, which require changing some of the options in order to work properly, see *Configuration requirements*.

All compile-time switches and their default values can be found in the file `USB_ConfDefaults.h`.

To change the default configuration of emUSB-Device, compile-time switches can be added to `USB_Conf.h`. Don't change the `USB_ConfDefaults.h` file for easy updates of emUSB-Device.

3.4.1 Compile-time switches for debugging

3.4.1.1 USB_DEBUG_LEVEL

Description

emUSB-Device can be configured to display debug messages and warnings to locate an error or potential problems. This can be useful for debugging. In a release (production) build of a target system, they are typically not required and should be switched off.

To output the messages, emUSB-Host uses the logging routines contained in `USB_ConfigIO.c` which can be customized.

`USB_DEBUG_LEVEL` can be set to the following values:

- 0 - Used for release builds. Includes no debug options.
- 1 - Used in debug builds to include support for "panic" checks.
- 2 - Used in debug builds to include warning, log messages and "panic" checks. This significantly increases the code size.

Definition

```
#define USB_DEBUG_LEVEL    0
```

3.4.1.2 USB_LOG_BUFFER_SIZE

Description

Maximum size of a debug / warning message (in characters) that can be output. A buffer of this size is created on the stack when a message is output.

Definition

```
#define USB_LOG_BUFFER_SIZE    100
```

3.4.2 Use of standard C-library functions

emUSB-Device calls some functions from the standard C-library. If the standard C-library should not be used, the following macros can be changed to call user-defined functions instead:

```
#define USBH_MEMCPY   memcpy
#define USBH_MEMSET   memset
#define USBH_MEMCMP   memcmp
#define USBH_MEMMOVE  memmove
#define USBH_STRLEN   strlen
#define USBH_STRCAT   strcat
#define USBH_STRRCHR  strrchr
#define USBH_STRNCPY  strncpy
#define USBH_STRCMP   strcmp
```

 © 2010-2026 SEGGER Microcontroller GmbH

3.4.3 General USB configuration

3.4.3.1 USB_SUPPORT_TRANSFER_ISO

Description

Must be set to 1 if the USB stack shall support isochronous transfers (e.g. for audio and video applications). If set to 0, all code that handles isochronous transfers is disabled, which may significantly reduce the code size of the USB stack.

Definition

```
#define USB_SUPPORT_TRANSFER_ISO    0
```

3.4.3.2 USB_SUPPORT_TEST_MODE

Description

USB test mode enable. This can be set to 1 to enable support for USB high-speed test mode. It is disabled by default to reduce memory footprint of the USB stack.

Definition

```
#define USB_SUPPORT_TEST_MODE    0
```

3.4.3.3 USB_NUM_EPS

Description

Maximum number of endpoints that can be used in the main stack. A table is stored in static memory with USB_NUM_EPS entries, each allocates 56 bytes.

Definition

```
#define USB_NUM_EPS    8u
```

3.4.3.4 USB_MAX_NUM_IF

Description

Maximum number of USB interfaces the device can support. Each USB class has one or more interfaces. For a USB composite device USB_MAX_NUM_IF must be ≥ the sum of the number of interfaces for all classes configured. Information about all interfaces are stored into a table, which allocates about USB_MAX_NUM_IF * 9 words in static memory (word = sizeof(void *)).

Definition

```
#define USB_MAX_NUM_IF    4u
```

3.4.3.5 USB_MAX_NUM_ALT_IF

Description

Maximum number of USB alternate settings the device can support. Some USB interfaces may have alternate settings to select different operating modes / configurations. USB_MAX_NUM_ALT_IF must be ≥ the sum of the number of all alternate settings of all interfaces of all classes configured. Information about all alternates settings are stored into a table, which allocates about USB_MAX_NUM_ALT_IF * 10 words in static memory (word = sizeof(void *)).

 © 2010-2026 SEGGER Microcontroller GmbH

Definition

```
#define USB_MAX_NUM_ALT_IF    2u
```

Number of interfaces / alternate settings for each USB class

USB class	Number of interfaces	Number of alternate settings
Bulk / Vendor	1	usually 0, but may be configured to 1
CDC-ACM	2	0
CDC-ECM	2	1
CDC-NCM	2	1
CCID	1	0
HID	1	0
Mass storage	1	0
MTP	1	0
Printer	1	0
DFU	0	1
MIDI	2	0
RNDIS	2	1
UVC	2	1
Audio	1 + Number of audio streams (IN/OUT)	At least one per audio stream, see *Design of audio interfaces*

3.4.3.6 USB_DESC_BUFFER_SIZE

Description

Size of the largest USB descriptor used in the device. Usually the configuration descriptor is the largest one. It consists of a header and the concatenation of configurations for all configured USB classes. The configuration descriptors are small for most classes, but for the audio or video class it may get large and require to increase the size of this buffer.

The value can be defined to 0. In that case dynamic allocation of the descriptor buffer is used, see `USBD_X_GetDescBuffer()`.

Definition

```
#define USB_DESC_BUFFER_SIZE    256
```

3.4.3.7 USBD_SUPPORT_PROFILE

Description

Enables USB API instrumentation for SystemView.

Definition

```
#define USBD_SUPPORT_PROFILE    0
```

3.4.3.8 USBD_OS_USE_USBD_X_INTERRUPT

Description

If set emUSB-Device will use the functions `USBD_X_EnableInterrupt()` and `USBD_X_DisableInterrupt()` instead of disabling/enabling the interrupts globally. Those functions only

 © 2010-2026 SEGGER Microcontroller GmbH

disable/enable the USB interrupt. The functions are MCU specific and must be defined in the corresponding USB_Config_*.c file.

Definition

```
#define USBD_OS_USE_USBD_X_INTERRUPT    0
```

3.4.3.9 USBD_OS_USE_ISR_FLAG

Description

Enable flag to indicate execution of USB interrupt. May be needed by some RTOS layer in combination with USBD_OS_USE_USBD_X_INTERRUPT=1 (experimental).

Definition

```
#define USBD_OS_USE_ISR_FLAG    0
```

 © 2010-2026 SEGGER Microcontroller GmbH

3.5 Security considerations

In order to achieve safe operation of the product containing emUSB-Device and avoid any vulnerabilities the following recommendations must be observed:

- All API functions must be used according to the function description in this manual. Functions not documented in this manual shall not be called by the application.
- When calling an API function, all parameter values must be in a valid range, as described in the function documentation.
- When calling an API function, all pointer parameters must point to a valid memory area that has an appropriate size and the required access right for the operation of the function. Especially read and write data buffers must have the capacity required for the data to be transferred. Pointer parameters must not be `NULL`, except if explicitly allowed in the function documentation.
- API functions may return status information (’success’ or ’error’). Error indications returned from API functions must not be ignored by the application. Any data that may be returned by the functions must not be processed when the function returns an error status.
- Functions that are provided by the application and called by emUSB-Device (callback functions) must be implemented as described in the documentation of the callback function.
- While emUSB-Device is running, internal data structures of emUSB-Device must never be accessed directly by the application and should never be modified: Neither intentionally nor inadvertently by out-of-bounds accesses or any hardware operation.
- In any context where functions of emUSB-Device are called, the current stack space must be large enough to run the function (and all sub-functions) without a stack overflow.
- While emUSB-Device is running, the application must not access a peripheral that is controlled by emUSB-Device, except in the initialization phase, in order to set up and configure that peripheral.
- Compile-time configuration options must be set in a valid range as documented in *Compile-time configuration*, when changed from the default setting.
- A complete test (full coverage) of the application using emUSB-Device should be run in debug configuration (compiled with ’#define `USB_DEBUG_LEVEL` 2’) capturing all warning output from emUSB-Device and all calls to `USB_OS_Panic()`, in order to detect and fix configuration errors.
- Correct operation of emUSB-Device can be expected only, if the underlying hardware operates correctly, as documented by the silicon vendor.

 © 2010-2026 SEGGER Microcontroller GmbH

3.6 Host OS specifics

3.6.1 Windows registry

The Windows registry is a database which stores settings for the operating system. The relevant aspect of the Windows registry in regard to USB development is the fact that Windows stores information about connected USB devices into the registry. Normally, Windows stores the Vendor and Product ID pair together with the USB configuration of that particular device in the registry. During USB development, this can have negative effects because if you, the developer, change the USB configuration of a device, Windows will still have the old USB configuration saved in the registry. While the USB device is functioning perfectly fine, the old registry entry can result in the device not being properly recognized by Windows.

This issue is especially prevalent when developing a USB Audio device.

3.6.1.1 Cleaning the Windows registry

The easiest way is to use a tool such as Uwe Sieber's "Device Cleanup Tool": https://www.uwe-sieber.de/misc_tools_e.html This tool allows any **not connected** devices to be removed from the registry.

Alternatively, the registry can be cleaned by hand using the Windows registry editor.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 4

USB Core

This chapter describes the basic functions of the USB Core.

 © 2010-2026 SEGGER Microcontroller GmbH

4.1 Overview

This chapter describes the functions of the core layer of emUSB-Device. These functions are required for all USB class drivers and the unclassified bulk communication component.

General information

To communicate with the host, the example applications include a USB-specific header `USB.h`. This file contains API functions to communicate with the USB host through the USB Core driver.

Every application using USB Core must perform the following steps:

1. Initialize the USB stack. To initialize the USB stack, `USBD_Init()` has to be called. `USBD_Init()` performs the low-level initialization of the USB stack and calls `USBD_X_Config()` to add a driver to the USB stack.
2. Add communication endpoints. You have to add the required endpoints with the compatible transfer type for the desired interface before you can use any of the USB class drivers or the unclassified bulk communication component. For the emUSB-Device bulk component, refer to `USB_BULK_INIT_DATA` on page 149 for information about the initialization structure that is required when you want to add a bulk interface. For the emUSB-Device MSD component, refer to *`USB_MSD_INIT_DATA`* on page 271 and *`USB_MSD_INST_DATA`* on page 273 for information about the initialization structures that are required when you want to add an MSD interface. For the emUSB-Device CDC component, refer to *`USB_CDC_INIT_DATA`* on page 414 for information about the initialization structure that is required when you want to add a CDC interface. For the emUSB-Device HID component, refer to *`USB_HID_INIT_DATA`* on page 446 for information about the initialization structure that is required when you want to add a HID interface.
3. Provide device information using `USBD_SetDeviceInfo()`.
4. Start the USB stack. Call `USBD_Start()` to start the USB stack.

Example applications for every supported USB class and the unclassified bulk component are supplied. We recommend using one of these examples as a starting point for your own application. All examples are supplied in the `\Application\` directory.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2 Target API

This section describes the functions that can be used by the target application.

Function	Description
USB basic functions	
USBD_Init()	Initializes the USB device with its settings.
USBD_Start()	Starts the emUSB-Device Core.
USBD_GetVersion()	Returns the version of the stack.
USBD_GetState()	Returns the state of the USB device.
USBD_IsConfigured()	Checks if the USB device is initialized and ready.
USBD_GetSpeed()	Returns the current connection speed.
USBD_GetDeviceState()	Returns the state of the USB device, set by the host (except USB_DEVSTAT_SELF_POWERED, which is configured by the device, see USBD_SetMaxPower()).
USBD_Stop()	Stops the USB communication.
USBD_DeInit()	De-initialize the complete USB stack.
USB configuration functions	
USBD_AddDriver()	Adds a USB device driver to the USB stack.
USBD_SetISREnableFunc()	Register function to enable USB interrupts.
USBD_SetAttachFunc()	Sets a function to perform hardware-specific actions to attach USB.
USBD_AddEP()	Returns an endpoint "handle" that can be used for the desired USB interface.
USBD_AddEPEx()	Returns an endpoint "handle" that can be used for the desired USB interface.
USBD_SetDeviceInfo()	Sets all information used during device enumeration.
USBD_SetClassRequestHook()	Sets a callback function that is called when a setup class request is sent from the host to the specified interface index.
USBD_SetVendorRequestHook()	Sets a callback function that is called when a setup vendor request is sent from the host to the specified interface index.
USBD_SetIsSelfPowered()	Sets whether the device is self-powered or not.
USBD_SetMaxPower()	Sets the maximum power consumption reported to the host during enumeration.
USBD_SetOnEvent()	Sets a callback function for an endpoint that will be called on every RX or TX event for that endpoint.
USBD_RemoveOnEvent()	Removes a callback function which was added via USBD_SetOnEvent from the callback list.
USBD_SetOnRxEP0Hook()	Sets a callback when data are received in the data stage of the setup request.
USBD_SetOnRXHookEP()	Sets a callback whenever data are received from a given endpoint handle.

 © 2010-2026 SEGGER Microcontroller GmbH

Function	Description
USBD_SetOnSetup()	Sets a callback function that is called when any setup request is sent from the host.
USBD_SetOnSetupHook()	Obsolete, use USBD_SetOnSetup().
USBD_SetOnSOF()	Installs a function that will be called, when a SOF was received from the host.
USBD_RemoveOnSOF()	Removes a callback function which was added via USBD_SetOnSOF() from the call-back list.
USBD_WriteEP0FromISR()	Writes data to EP0 (control endpoint).
USBD_EnableIAD()	Enables combination of multi-interface device classes with single-interface classes or other multi-interface classes.
USBD_SetCacheConfig()	Configures cache related functionality that might be required by the stack for cache handling in drivers.
USBD_RegisterSCHook()	Sets a callback function that will be called on every state change of the USB device.
USBD_AssignMemory()	Assigns an area of RAM to be used for the endpoint buffers and transfer descriptors by the USB driver.
USBD_UseV210()	Enable use of USB V2.10 specification revision.
USBD_SetBESLValues()	Set recommended BESL (Best Effort Service Latency) values to be used in the BOS descriptor when using LPM (Link Power Management).
USBD_SetOnLPMChange()	Sets a call back to report LPM transition on the USB lines (L0 <-> L1).
USBD_SetLPMResponse()	Defines the behavior of the device on LPM requests from the host.
USBD_EnableSuperSpeed()	Enable SuperSpeed in the USB stack.
USBD_SetWebUSBInfo()	For WebUSB capable USB devices this function may be called before USBD_Start() to enable WebUSB specific descriptors.
USBD_SetCheckAddress()	Installs a function that checks if an address can be used for DMA transfers.
USBD_SetGetStringHook()	Sets a call to determine the string of a specified string index.
USBD_X_GetDescBuffer()	User provided function to allocate a buffer for the USB stack to construct USB descriptors requested by the host.
USB I/O functions	
USBD_Read()	Reads data from the host.
USBD_ReadOverlapped()	Reads data from the host asynchronously.
USBD_Receive()	Reads data from the host.
USBD_ReceivePoll()	Reads data from the host.
USBD_ReadAsync()	Reads data from the host asynchronously.
USBD_Write()	Writes data to the host.
USBD_WriteAsync()	Sends data to the host asynchronously.

 © 2010-2026 SEGGER Microcontroller GmbH

Function	Description
USBD_CancelIO()	Cancel any read or write operation.
USBD_WaitForEndOfTransferEx()	Waits until the current transfer on a particular EP completes.
USBD_WaitForTXReady()	Waits (blocking) until the TX queue can accept another data packet.
USBD_GetNumBytesInBuffer()	Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer.
USBD_GetNumBytesRemToRead()	This function is to be used in combination with USBD_ReadOverlapped().
USBD_GetNumBytesRemToWrite()	This function is to be used in combination with a non-blocking call to USBD_Write().
USBD_StallEP()	Stalls an endpoint.
USB RemoteWakeUp functions	
USBD_SetAllowRemoteWakeUp()	Allows the device to publish that remote wake is available.
USBD_DoRemoteWakeup()	Performs a remote wakeup in order to wake up the host from the standby/suspend state.
Data structures	
USB_ASYNC_IO_CONTEXT	Contains information for asynchronous transfers.
USB_SETUP_PACKET	Structure containing a USB setup packet.
SEGGER_CACHE_CONFIG	Used to pass cache configuration and callback function pointers to the stack.
USB_CHECK_ADDRESS_FUNC	Checks if an address can be used for DMA transfers.
USB_WEBUSB_INFO	Information that may be provided by the application for WebUSB capable USB devices.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.1 USB basic functions

4.2.1.1 USBD_GetState()

Description

Returns the state of the USB device.

Prototype

```
unsigned USBD_GetState(void);
```

Return value

A bitwise combination of the USB state flags:

USB_STAT_ATTACHED	Device is attached. (Note 1)
USB_STAT_READY	Device is ready. (Note 2)
USB_STAT_ADDRESSED	Device is addressed. (Note 3)
USB_STAT_CONFIGURED	Device is configured. (Note 4)
USB_STAT_SUSPENDED	Device is suspended. (Note 5)

Additional information

A USB device has several possible states. Some of these states are visible to the USB and the host, while others are internal to the USB device. Refer to Universal Serial Bus Specification, Revision 2.0, Chapter 9 for detailed information.

Notes

(1) Attached in a USB-specification sense of the word does not mean that the device is physically connected to the host via a USB cable, it only means that the pull-up resistor on the device side is connected. The status can be “attached” regardless of whether the device is connected to a host or not. This state can normally be ignored.

(2) Ready denotes the USB controller state, the controller is “ready” after a bus reset. This state can normally be ignored.

(3) A device is in an addressed state after it receives a valid (non-zero) USB address from the USB host. This state can normally be ignored.

(4) When a device is “configured” the enumeration of the device has been successfully completed and the host can communicate with the device.

(5) Suspend is set when the device is physically disconnected from the host or when the USB host suspends the connected device.

Mapping of the state value returned by `USBD_GetState()` to the USB states described in “Universal Serial Bus Specification Revision 2.0”, chapter 9.1:

Return value of `USBD_GetState()`	USB state
`0x10` = 10000_B	Attached
`0x11` = 10001_B	Powered + Suspended
`0x18` = 11000_B	Default
`0x19` = 11001_B	Default + Suspended
`0x1C` = 11100_B	Address
`0x1D` = 11101_B	Address + Suspended
`0x1E` = 11110_B	Configured
`0x1F` = 11111_B	Configured + Suspended
Other values should not occur	

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.1.2 USBD_GetSpeed()

Description

Returns the current connection speed.

Prototype

```
int USBD_GetSpeed(void);
```

Return value

USB_SPEED_NONE	Unknown speed.
USB_SPEED_FS	Full-speed.
USB_SPEED_HS	High-speed.
USB_SPEED_SS	SuperSpeed.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.1.3 USBD_GetDeviceState()

Description

Returns the state of the USB device, set by the host (except `USB_DEVSTAT_SELF_POWERED`, which is configured by the device, see `USBD_SetMaxPower()`).

Prototype

```
unsigned USBD_GetDeviceState(void);
```

Return value

A bitwise combination of the USB device state flags:

`USB_DEVSTAT_SELF_POWERED`	Device is self-powered.
`USB_DEVSTAT_REMOTE_WAKEUP_ALLOWED`	Remote Wakeup is allowed.
`USB_DEVSTAT_U1_ENABLE`	Link power state U1 is enabled (SuperSpeed only).
`USB_DEVSTAT_U2_ENABLE`	Link power state U2 is enabled (SuperSpeed only).
`USB_DEVSTAT_LPM_ENABLE`	Link power management is enabled (SuperSpeed only).

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.1.4 USBD_Init()

Description

Initializes the USB device with its settings.

Prototype

```
void USBD_Init(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.1.5 USBD_IsConfigured()

Description

Checks if the USB device is initialized and ready.

Prototype

```
char USBD_IsConfigured(void);
```

Return value

0	USB device is not configured.
1	USB device is configured.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.1.6 USBD_Start()

Description

Starts the emUSB-Device Core.

Prototype

```
void USBD_Start(void);
```

Additional information

This function should be called after configuring USB Core. It initiates a hardware attach and updates the endpoint configuration. When the USB cable is connected to the device, the host will start enumeration of the device.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.1.7 USBD_Stop()

Description

Stops the USB communication. This also makes sure that the device is detached from the HOST.

Prototype

```
void USBD_Stop(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.1.8 USBD_DeInit()

Description

De-initialize the complete USB stack.

Prototype

```
void USBD_DeInit(void);
```

Additional information

This function also calls `USBD_Stop()` internally.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.1.9 USBD_GetVersion()

Description

Returns the version of the stack.

Prototype

```
U32 USBD_GetVersion(void);
```

Return value

Format: Mmmrr; e.g: 32401 is 3.24.1

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2 USB configuration functions

4.2.2.1 USBD_AddDriver()

Description

Adds a USB device driver to the USB stack. This function should be called from within `USBD_X_Config()` which is implemented in `USB_Config_*.c`.

Prototype

```
void USBD_AddDriver(const USB_HW_DRIVER * pDriver);
```

Parameters

Parameter	Description
pDriver	Pointer to the driver API structure.

Additional information

To add the driver, use `USBD_AddDriver()` with the identifier of the compatible driver. Refer to the section “Available target USB drivers” in the `USB.h` header file for a list of supported devices and their valid identifiers.

Example

```
/*********************************************************************
*
*       USBD_X_Config
*/
void USBD_X_Config(void) {
  BSP_USB_Init();
  USB_DRIVER_LPC17xx_ConfigAddr(0x2008C000);  // USB controller of LPC1788
                                              // is located @ 0x2008C000
  USBD_AddDriver(&USB_Driver_NXPLPC17xx);
  USBD_SetISREnableFunc(_EnableISR, NULL, NULL);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.2 USBD_SetISREnableFunc()

Description

Register function to enable USB interrupts.

Prototype

```
void USBD_SetISREnableFunc(USB_ENABLE_ISR_FUNC * pfEnableISR);
```

Parameters

Parameter	Description
pfEnableISR	Pointer to the function to install the interrupt handler and enable the USB interrupt.

Additional information

This function must be called within the USBD_X_Config() function. See *Adding a driver to emUSB-Device* on page 747. The function pointer prototype is defined as follows:

```
typedef void  USB_ENABLE_ISR_FUNC (USB_ISR_HANDLER * pfISRHandler);
```

Example

See USBD_AddDriver().

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.3 USBD_SetAttachFunc()

Description

Sets a function to perform hardware-specific actions to attach USB.

Prototype

```
void USBD_SetAttachFunc(USB_ATTACH_FUNC * pfAttach);
```

Parameters

Parameter	Description
pfAttach	Pointer to the attach function.

Additional information

This function must be called within the USBD_X_Config() function. See *Adding a driver to emUSB-Device* on page 747. The function pointer prototypes are defined as follows:

```
typedef void  USB_ATTACH_FUNC (void);
```

Example

See *USBD_X_Config()* on page 747.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.4 USBD_AddEP()

Description

Returns an endpoint “handle” that can be used for the desired USB interface.

Prototype

```
unsigned USBD_AddEP(U8          InDir,
                    U8          TransferType,
                    U16         Interval,
                    U8        * pBuffer,
                    unsigned    BufferSize);
```

Parameters

Parameter	Description
InDir	Specifies the direction of the desired endpoint. • USB_DIR_IN • USB_DIR_OUT
TransferType	Specifies the transfer type of the endpoint. The following values are allowed: • USB_TRANSFER_TYPE_BULK • USB_TRANSFER_TYPE_INT ISO endpoints must be created using USBD_AddEPEx().
Interval	Specifies the interval measured in units of 125us (micro frames). This value should be zero for a bulk endpoint.
pBuffer	Pointer to a buffer that is used for OUT-transactions. For IN-endpoints this parameter must be NULL.
BufferSize	Size of the buffer (OUT endpoints only). Must be a multiple of the maximum packet size.

Return value

> 0	A valid endpoint handle is returned.
= 0	Error.

Additional information

The Interval parameter specifies the frequency in which the endpoint should be polled for information by the host. It must be specified in units of 125 us.

Depending on the actual speed of the device during enumeration, the USB stack converts the interval to the correct value required for the endpoint descriptor according to the USB specification (into milliseconds for low/full-speed, into 125 us for high-speed).

For endpoints of type USB_TRANSFER_TYPE_BULK the value is ignored and should be set to 0.

This function must be called after USBD_Init() and before USBD_Start().

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.5 USBD_AddEPEx()

Description

Returns an endpoint “handle” that can be used for the desired USB interface.

Prototype

```
unsigned USBD_AddEPEx(const USB_ADD_EP_INFO * pInfo,
                      U8                    * pBuffer,
                      unsigned                BufferSize);
```

Parameters

Parameter	Description
pInfo	Pointer to a structure of type USB_ADD_EP_INFO.
pBuffer	Pointer to an endpoint buffer that is used for OUT-transactions. For IN-endpoints or ISO endpoints this parameter should be NULL.
BufferSize	Size of the endpoint buffer (OUT endpoints only). Must be ≥ the maximum packet size of the endpoint. For IN-endpoints or ISO endpoints this parameter should be 0.

Return value

> 0 A valid endpoint handle is returned.
= 0 Error.

Additional information

This function must be called after USBD_Init() and before USBD_Start().

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.6 USBD_SetDeviceInfo()

Description

Sets all information used during device enumeration.

Prototype

```
void USBD_SetDeviceInfo(const USB_DEVICE_INFO * pDeviceInfo);
```

Parameters

Parameter	Description
pDeviceInfo	Pointer to a structure containing the device information. Must point to static data that is not changed while the stack is running.

Additional information

See *USB_DEVICE_INFO* on page 53 for a description of the structure.

Example

See *USB_DEVICE_INFO* on page 53.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.7 USBD_SetClassRequestHook()

Description

Sets a callback function that is called when a setup class request is sent from the host to the specified interface index.

Prototype

```
void USBD_SetClassRequestHook(unsigned               InterfaceNum,
                              USB_ON_CLASS_REQUEST * pfOnClassRequest);
```

Parameters

Parameter	Description
InterfaceNum	Interface index for setting the class request callback.
pfOnClassRequest	Pointer to the callback.

Additional information

Note that the callback will be called within an ISR, therefore it should never block. If it is necessary to send data from the callback function through endpoint 0, use the function USBD_WriteEP0FromISR().

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.8 USBD_SetVendorRequestHook()

Description

Sets a callback function that is called when a setup vendor request is sent from the host to the specified interface index.

Prototype

```
void USBD_SetVendorRequestHook(unsigned                InterfaceNum,
                               USB_ON_CLASS_REQUEST * pfOnVendorRequest);
```

Parameters

Parameter	Description
InterfaceNum	Interface index for setting the class request callback.
pfOnVendorRequest	Pointer to the callback.

Additional information

Note that the callback will be called within an ISR, therefore it should never block. If it is necessary to send data from the callback function through endpoint 0, use the function USBD_WriteEP0FromISR().

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.9 USBD_SetIsSelfPowered()

Description

Sets whether the device is self-powered or not. Obsolete function, please use `USBD_SetMaxPower()`.

Prototype

```
void USBD_SetIsSelfPowered(U8 IsSelfPowered);
```

Parameters

Parameter	Description
`IsSelfPowered`	• 0 - Device is not self-powered. • 1 - Device is self-powered.

Additional information

This function has to be called before `USBD_Start()`, as it will specify if the device is self-powered or not. The default value is 0 (not self-powered).

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.10 USBD_SetMaxPower()

Description

Sets the maximum power consumption reported to the host during enumeration. This function also sets whether the device is self-powered (`MaxPower` = 0) or not.

Prototype

```
void USBD_SetMaxPower(unsigned MaxPower);
```

Parameters

Parameter	Description
`MaxPower`	Maximum power consumption of the device given in mA. `MaxPower` shall be in range between 0mA - 500mA, for SuperSpeed devices up to 900mA.

Additional information

This function shall be called before `USBD_Start()`, as it will specify how much power the device will consume from the host. If this function is not called, a default value of 100 mA will be used.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.11 USBD_SetOnEvent()

Description

Sets a callback function for an endpoint that will be called on every RX or TX event for that endpoint.

Prototype

```
void USBD_SetOnEvent(unsigned                   EPIndex,
                     USB_EVENT_CALLBACK       * pEventCb,
                     USB_EVENT_CALLBACK_FUNC  * pfEventCb,
                     void                     * pContext);
```

Parameters

Parameter	Description
EPIndex	Endpoint index returned by USBD_AddEP().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure (will be initialized by this function).
pfEventCb	Pointer to the callback routine that will be called on every event on the USB endpoint.
pContext	A pointer which is used as parameter for the callback function.

Additional information

The USB_EVENT_CALLBACK structure is private to the USB stack. It will be initialized by USBD_SetOnEvent(). The USB stack keeps track of all event callback functions using a linked list. The USB_EVENT_CALLBACK structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a read or write operation was started for the endpoint using one of the USBD_Read…() or USBD_Write…() functions.

Additional information

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
Events	A bit mask indicating which events occurred on the endpoint.
pContext	The pointer which was provided to the USBD_SetOnEvent() function.

Note that the callback function will be called within an ISR; therefore, it should never block. The first parameter to the callback function will contain a bit mask for all events that triggered the call:

Event	Description
USB_EVENT_DATA_READ	Some data was received from the host on the endpoint.
USB_EVENT_DATA_SEND	Some data was sent to the host, so that (part of) the user write buffer may be reused by the application.
USB_EVENT_DATA_ACKED	Some data was acknowledged by the host.
USB_EVENT_READ_COMPLETE	The last read operation was completed.

 © 2010-2026 SEGGER Microcontroller GmbH

Event	Description
USB_EVENT_READ_ABORT	A read transfer was aborted.
USB_EVENT_WRITE_ABORT	A write transfer was aborted.
USB_EVENT_WRITE_COMPLETE	All write operations were completed.

Example

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
  if ((Events & USB_EVENT_DATA_SEND) != 0 &&
        // Check for last write transfer to be completed.
        USBD_GetNumBytesRemToWrite(EPIndex) == 0) {
            <.. prepare next data for writing..>
            // Send next packet of data.
            r = USBD_Write(EPIndex, &ac[0], 200, 0, -1);
            if (r < 0) {
              <.. error handling..>
            }
   }
}
// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBD_SetOnEvent(EPIndex, &_usb_callback, _OnEvent, NULL);
// Send the first packet of data using an asynchronous write operation.
r = USBD_Write(EPIndex, &ac[0], 200, 0, -1);
if (r < 0) {
  <.. error handling..>
}
<.. do anything else here while the whole data is sent..>
```

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.12 USBD_RemoveOnEvent()

Description

Removes a callback function which was added via `USBD_SetOnEvent` from the callback list.

Prototype

```
void USBD_RemoveOnEvent(      unsigned             EPIndex,
                        const USB_EVENT_CALLBACK * pEventCb);
```

Parameters

Parameter	Description
`EPIndex`	Endpoint index returned by `USBD_AddEP()`.
`pEventCb`	Pointer to a `USB_EVENT_CALLBACK` structure which was used with `USBD_SetOnEvent`.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.13 USBD_SetOnRxEP0Hook()

Description

Sets a callback when data are received in the data stage of the setup request.

Prototype

```
void USBD_SetOnRxEP0Hook(USB_RXEP_HOOK      * pHook,
                         USB_ON_RX_FUNC_V2 * pfOnRx,
                         void              * pContext);
```

Parameters

Parameter	Description
pHook	Pointer to static USB_RXEP_HOOK structure.
pfOnRx	Pointer to OnRx callback.
pContext	User defined context that is passed to the callback function.

Additional information

The USB_RXEP_HOOK structure is private to the USB stack. It will be initialized by USBD_SetOnRxEP0Hook(). The USB stack keeps track of all 'EP0 received DATA' callback functions using a linked list. The USB_RXEP_HOOK structure will be included into this linked list and must reside in static memory.

Please note that this function can be called multiple times from different classes in order to check the data.

Note that the callback will be called within an ISR, therefore it should never block. If it is necessary to send data from the callback function through endpoint 0, use the function USBD_WriteEP0FromISR().

USB_ON_RX_FUNC_V2 is defined as follows:

```
typedef void USB_ON_RX_FUNC_V2(const U8 * pData, unsigned NumBytes, void *pContext);
```

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.14 USBD_SetOnRXHookEP()

Description

Sets a callback whenever data are received from a given endpoint handle. The callback function is called within the interrupt context and must not block.

Prototype

```
void USBD_SetOnRXHookEP(unsigned          EPIndex,
                        USB_ON_RX_FUNC * pfOnRx);
```

Parameters

Parameter	Description
EPIndex	Any valid endpoint handle > 0.
pfOnRx	Pointer to the callback.

USB_ON_RX_FUNC is defined as follows:

```
typedef void USB_ON_RX_FUNC(const U8 * pData, unsigned NumBytes);
```

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.15 USBD_SetOnSetup()

Description

Sets a callback function that is called when any setup request is sent from the host.

Prototype

```
void USBD_SetOnSetup(USB_SETUP_HOOK * pHook,
                     USB_ON_SETUP   * pfOnSetup);
```

Parameters

Parameter	Description
pHook	Pointer to a USB_SETUP_HOOK structure (will be initialized by this function).
pfOnSetup	Pointer to the callback function.

Additional information

The USB_SETUP_HOOK structure is private to the USB stack. It will be initialized by USBD_SetOnSetup(). The USB stack keeps track of all setup callback functions using a linked list. The USB_SETUP_HOOK structure will be included into this linked list and must reside in static memory.

Note that the callback will be called within an ISR, therefore it should never block. If it is necessary to send data from the callback function through endpoint 0, use the function USBD_WriteEP0FromISR().

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.16 USBD_SetOnSetupHook()

Description

Obsolete, use `USBD_SetOnSetup()`. Sets a callback function that is called when any setup request is sent from the host.

Prototype

```
void USBD_SetOnSetupHook(unsigned       InterfaceNum,
                         USB_ON_SETUP * pfOnSetup);
```

Parameters

Parameter	Description
`InterfaceNum`	Interface index for setting the setup request callback.
`pfOnSetup`	Pointer to the callback function.

Additional information

Note that the callback will be called within an ISR, therefore it should never block. If it is necessary to send data from the callback function through endpoint 0, use the function `USBD_WriteEP0FromISR()`.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.17 USBD_SetOnSOF()

Description

Installs a function that will be called, when a SOF was received from the host. The callback function is called within the interrupt context and must not block.

Prototype

```
int USBD_SetOnSOF(void                    (*pfSOFCallback)(void * pContext),
                  U16                     Interval,
                  void                  * pContext,
                  USB_SOF_CALLBACK_HOOK * pHook);
```

Parameters

Parameter	Description
pfSOFCallback	Pointer to the callback function.
Interval	Function will be called every time a number of 'Interval' SOFs were received.
pContext	A pointer which is used as parameter for the callback function.
pHook	Pointer to a USB_SOF_CALLBACK_HOOK structure (will be initialized by this function).

Return value

= 0 Callback function successfully installed.
≠ 0 SOF callback not supported by the driver.

Additional information

The USB_EVENT_CALLBACK structure is private to the USB stack. It will be initialized by USBD_SetOnEvent(). The USB stack keeps track of all event callback functions using a linked list. The USB_EVENT_CALLBACK structure will be included into this linked list and must reside in static memory.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.18 USBD_RemoveOnSOF()

Description

Removes a callback function which was added via `USBD_SetOnSOF()` from the callback list.

Prototype

```
void USBD_RemoveOnSOF(const USB_SOF_CALLBACK_HOOK * pHook);
```

Parameters

Parameter	Description
pHook	Pointer to a `USB_SOF_CALLBACK_HOOK` structure which was installed using `USBD_SetOnSOF()`.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.19 USBD_WriteEP0FromISR()

Description

Writes data to EP0 (control endpoint). This function may be called in an interrupt context.

Prototype

```
void USBD_WriteEP0FromISR(const void     * pData,
                          unsigned   NumBytes,
                          I8         Send0PacketIfRequired);
```

Parameters

Parameter	Description
pData	Data that should be written.
NumBytes	Number of bytes to write.
Send0PacketIfRequired	Specifies that a zero-length packet should be sent when the last data packet to the host is a multiple of MaxPacketSize. Normally, MaxPacketSize for control mode transfers is 64 bytes.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.20 USBD_EnableIAD()

Description

Enables combination of multi-interface device classes with single-interface classes or other multi-interface classes.

Prototype

```
void USBD_EnableIAD(void);
```

Additional information

Simple device classes such as HID and MSD or BULK use only one interface descriptor to describe the class. The interface descriptor also contains the device class code. Multi-interface device classes, such as CDC, Audio, MIDI use more than one interface descriptor to describe the class. The device class code will then be written into the device descriptor. It may be possible to add an interface which does not belong to a multi-interface class, but it may not be correctly recognized by the host, this is not standardized and depends on the host. In order to allow this, a new descriptor type was introduced:

IAD (Interface Association Descriptor), this descriptor will encapsulate the multi-interface class into this IA descriptor, so that it will be seen as one single interface and will then allow adding other device classes.

If you intend to use a multi-interface component with any other component, please call `USBD_EnableIAD()` before adding the multi-interface component through `USBD_*_Add()`.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.21 USBD_SetCacheConfig()

Description

Configures cache related functionality that might be required by the stack for cache handling in drivers.

Prototype

```
void USBD_SetCacheConfig(const SEGGER_CACHE_CONFIG * pConfig,
                         unsigned                    ConfSize);
```

Parameters

Parameter	Description
pConfig	Pointer to an element of SEGGER_CACHE_CONFIG .
ConfSize	Size of the passed structure in case library and header size of the structure differs.

Additional information

This function has to be called in USBD_X_Config().

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.22 USBD_RegisterSCHook()

Description

Sets a callback function that will be called on every state change of the USB device.

Prototype

```
int USBD_RegisterSCHook(USB_HOOK                 * pHook,
                        USB_STATE_CALLBACK_FUNC * pfStateCb,
                        void                     * pContext);
```

Parameters

Parameter	Description
pHook	Pointer to a USB_HOOK structure (will be initialized by this function).
pfStateCb	Pointer to the callback routine that will be called on every state change.
pContext	A pointer which is used as parameter for the callback function.

Return value

0 OK.
1 Error, specified hook already exists.

Additional information

The USB_HOOK structure is private to the USB stack. It will be initialized by USBD_RegisterSCHook(). The USB stack keeps track of all state change callback functions using a linked list. The USB_HOOK structure will be included into this linked list and must reside in static memory.

Note that the callback function will be called within an ISR, therefore it should never block.

Example

```
// The callback function.
static void _OnStateChange(void *pContext, U8 NewState) {
  if ((NewState & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED)) == USB_STAT_CONFIGURED) {
    // Device is enumerated
    } else {
    // Device not enumerated
  }
}
// Main program.
static USB_HOOK Hook;

USBD_Init();
...
USBD_RegisterSCHook(&Hook, _OnStateChange, NULL);
...
USBD_Start();
```

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.23 USBD_AssignMemory()

Description

Assigns an area of RAM to be used for the endpoint buffers and transfer descriptors by the USB driver. This function should be called from within the `USBD_X_Config()` function after installing the driver with `USBD_AddDriver()`. Not all drivers support this function.

If the driver uses DMA, the USB controller must have DMA access to this area. For some drivers, the memory should be aligned to a given boundary. If not aligned, the driver will increase the start address and reduce the size of the area to achieve proper alignment. This results in wasting of RAM and may cause the driver to run out of memory.

Prototype

```
void USBD_AssignMemory(void * pMem,
                       U32    MemSize);
```

Parameters

Parameter	Description
`pMem`	Pointer to the start of the RAM area to be used by the driver.
`MemSize`	Size of the RAM area in bytes.

Additional information

If the memory is not sufficient for the class and endpoint configuration, the USB driver will run into the `USB_OS_Panic()` function during initialization, if compiled for DEBUG mode (`USB_DEBUG_LEVEL` > 0). After successful initialization, the driver will usually issue a `USB_LOG()` message to report, how many bytes of the assigned memory are not used. The size of the memory area may then be adjusted.

Information on how to calculate the size of the endpoint buffer memory and about any alignment requirements can be found in *Device driver specifics* on page 751.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.24 USBD_UseV210()

Description

Enable use of USB V2.10 specification revision. Must be called in `USBD_X_Config()`. It enables providing a BOS descriptor to the host and also enables link power management (LPM), if supported by the driver and the USB controller.

Prototype

```
void USBD_UseV210(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.25 USBD_SetBESLValues()

Description

Set recommended BESL (Best Effort Service Latency) values to be used in the BOS descriptor when using LPM (Link Power Management). See "Errata for USB 2.0 ECN: Link Power Management (LPM) - 7/2007" from usb.org for an explanation of these values. Calling this function has no effect, if LPM is not enabled (see `USBD_UseV210()`) or not supported by the driver or USB controller.

Prototype

```
void USBD_SetBESLValues(int BaselineBESL,
                        int DeepBESL);
```

Parameters

Parameter	Description
`BaselineBESL`	Recommended Baseline BESL value. Must be in range -1 to 15. A value of -1 means, no BESL value is stored in the BOS descriptor (the default). Values of 0,1,…,14,15 specify a BESL of 125us,150us,…,9000us,10000us respectively (see LPM document from usb.org).
`DeepBESL`	Recommended Deep BESL value. Must be in range -1 to 15 (see above).

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.26 USBD_SetOnLPMChange()

Description

Sets a call back to report LPM transition on the USB lines (L0 <-> L1).

Prototype

```
void USBD_SetOnLPMChange(USB_ON_LPM_CHANGE * pfOnLPMChange);
```

Parameters

Parameter	Description
pfOnLPMChange	Pointer to callback.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.27 USBD_SetLPMResponse()

Description

Defines the behavior of the device on LPM requests from the host. Calling this function has no effect, if LPM is not enabled (see `USBD_UseV210()`) or not supported by the driver or USB controller.

Prototype

```
void USBD_SetLPMResponse(U8 Response);
```

Parameters

Parameter	Description
`Response`	• 0 - LPM requests are rejected (NYET). • 1 - LPM requests are acknowledged.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.28 USBD_EnableSuperSpeed()

Description

Enable SuperSpeed in the USB stack. Must be called in `USBD_X_Config()`. If the USB driver or USB controller does not support SuperSpeed, calling this function has no effect.

Prototype

```
void USBD_EnableSuperSpeed(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.29 USBD_SetWebUSBInfo()

Description

For WebUSB capable USB devices this function may be called before `USBD_Start()` to enable WebUSB specific descriptors. This function can be used only, if the USB controller supports USB 2.1 compatibility, especially link power management (LPM).

Prototype

```
void USBD_SetWebUSBInfo(const USB_WEBUSB_INFO * pWebUSBInfo);
```

Parameters

Parameter	Description
`pWebUSBInfo`	Pointer to a structure containing the device information. Must point to static data that is not changed while the stack is running.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.30 USBD_SetCheckAddress()

Description

Installs a function that checks if an address can be used for DMA transfers. Installed function must return 0, if DMA access is allowed for the given address, any value ≠ 0 otherwise.

Prototype

```
void USBD_SetCheckAddress(USB_CHECK_ADDRESS_FUNC * pfCheckValidDMAAddress);
```

Parameters

Parameter	Description
pfCheckValidDMAAddress	Pointer to the function.

Additional information

If the function reports a memory region not valid for DMA, the driver uses a temporary transfer buffer to copy data to and from this area.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.31 USBD_SetGetStringHook()

Description

Sets a call to determine the string of a specified string index.

Prototype

```
void USBD_SetGetStringHook(USB_GET_STRING_DESC_HOOK * pHook,
                           USB_GET_STRING_FUNC      * pfOnGetString);
```

Parameters

Parameter	Description
pHook	Pointer to static USB_GET_STRING_DESC_HOOK structure.
pfOnGetString	Pointer to GetString callback.

Additional information

The USB_GET_STRING_DESC_HOOK structure is private to the USB stack. It will be initialized by USBD_SetGetStringHook(). The USB stack keeps track of all 'GetString' callback functions using a linked list. The USB_GET_STRING_DESC_HOOK structure will be included into this linked list and must reside in static memory.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.2.32 USBD_X_GetDescBuffer()

Description

User provided function to allocate a buffer for the USB stack to construct USB descriptors requested by the host.

This function is used only if the compiler switch *USB_DESC_BUFFER_SIZE* is set to 0. If *USB_DESC_BUFFER_SIZE* has a positive value a static buffer is allocated and this function is not called.

This function is called during each USB setup "get descriptor" request. The buffer returned by this function is used until this setup request is finished. The functions is called in interrupt context and must not block. Especially malloc() can not be used.

Prototype

```
void *USBD_X_GetDescBuffer(unsigned * pBuffSize);
```

Parameters

Parameter	Description
pBuffSize	out The size of the provided buffer must be returned.

Return value

Pointer to the provided buffer.

Simple implementation:

```
void * USBD_X_GetDescBuffer(unsigned *pBuffSize) {
  static char DescBuffer[500];
  *pBuffSize = sizeof(DescBuffer);
  return DescBuffer;
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3 USB I/O functions

4.2.3.1 USBD_Read()

Description

Reads data from the host.

Prototype

```
int USBD_Read(unsigned   EPOut,
              void     * pData,
              unsigned   NumBytesReq,
              unsigned   Timeout);
```

Parameters

Parameter	Description
EPOut	Handle to an OUT endpoint returned by USBD_AddEP().
pData	Pointer to a buffer where the received data will be stored.
NumBytesReq	Number of bytes to read.
Timeout	Timeout given in milliseconds. A zero value results in an infinite timeout.

Return value

= NumBytes	Requested data was successfully read within the given timeout.
≥ 0 && < NumBytes	Timeout has occurred (Number of bytes read before timeout).
< 0	An error occurred.

Additional information

This function blocks the task until all data have been read or a timeout occurs. In case of a reset or a disconnect, USB_STATUS_ERROR is returned.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint that was provided via the USBD_AddEP() function. This data can be retrieved by a later call to USBD_Receive() or USBD_Read(). See also USBD_GetNumBytesInBuffer().

In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.2 USBD_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

```
int USBD_ReadOverlapped(unsigned   EPOut,
                        void     * pData,
                        unsigned   NumBytesReq);
```

Parameters

Parameter	Description
EPOut	Handle to an OUT endpoint returned by USBD_AddEP().
pData	Pointer to a buffer where the received data will be stored.
NumBytesReq	Number of bytes to read.

Return value

≥ 0 Number of bytes that have been read from the internal buffer (success).
= 0 No data was found in the internal buffer, read transfer started (success).
< 0 An error occurred.

Additional information

This function will not block the calling task. The read transfer will be initiated and the function returns immediately. In order to synchronize, USBD_WaitForEndOfTransfer() needs to be called.

Another synchronization method would be to periodically call USBD_GetNumBytesRemToRead() in order to see how many bytes still need to be received (this method is preferred when a non-blocking solution is necessary).

The read operation can be canceled using USBD_CancelIO().

The buffer pointed to by pData must be valid until the read operation is terminated.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.3 USBD_Receive()

Description

Reads data from the host. The function blocks until any data have been received. In contrast to USBD_Read(), this function does not wait for all of NumBytes to be received, but returns after the first packet has been received or after a timeout occurs. In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

Prototype

```
int USBD_Receive(unsigned   EPOut,
                 void     * pData,
                 unsigned   NumBytesReq,
                 int        Timeout);
```

Parameters

Parameter	Description
EPOut	Handle to an OUT endpoint returned by USBD_AddEP().
pData	Pointer to a buffer where the received data will be stored.
NumBytesReq	Number of bytes to read.
Timeout	Timeout given in milliseconds. A zero value results in an infinite timeout. If Timeout is -1, the function never blocks and only reads data from the internal endpoint buffer.

Return value

> 0	Number of bytes that have been read within the given timeout.
= 0	A timeout occurred (if Timeout > 0), zero packet received (not every controller supports this!), no data in buffer (if Timeout < 0) or the target was disconnected during the function call and no data was read so far.
< 0	An error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling USBD_Receive() will return as much data as is currently available, up to the size of the buffer specified within the specified timeout. This function also returns when the target is disconnected from the host or when a USB reset occurs during the function call; it will then return the number of bytes read so far. If the target was disconnected before this function was called, it returns USB_STATUS_ERROR.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint that was provided via USBD_AddEP(). This data can be retrieved by a later call to USBD_Receive() / USBD_Read(). See also USBD_GetNumBytesInBuffer().

A call of USBD_Receive(EPOut, NULL, 0, -1) can be used to trigger an asynchronous read that stores the data into the internal buffer.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.4 USBD_ReceivePoll()

Description

Reads data from the host. The function blocks until any data have been received. In contrast to USBD_Read(), this function does not wait for all of NumBytes to be received, but returns after the first packet has been received or after a timeout occurs. In contrast to USBD_Receive(), this function will continue the read transfer asynchronously in case of a timeout.

Prototype

```
int USBD_ReceivePoll(unsigned   EPOut,
                     void     * pData,
                     unsigned   NumBytesReq,
                     unsigned   Timeout);
```

Parameters

Parameter	Description
EPOut	Handle to an OUT endpoint returned by USBD_AddEP().
pData	Pointer to a buffer where the received data will be stored.
NumBytesReq	Number of bytes to read.
Timeout	Timeout given in milliseconds. A zero value results in an infinite timeout.

Return value

> 0	Number of bytes that have been read within the given timeout.
= 0	A timeout occurred (if Timeout > 0) or a zero packet received (not every controller supports this!), or the target was disconnected during the function call and no data was read so far.
< 0	An error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling USBD_ReceivePoll() will return as much data as is currently available, up to the size of the buffer specified within the specified timeout. This function also returns when the target is disconnected from the host or when a USB reset occurs during the function call; it will then return the number of bytes read so far. If the target was disconnected before this function was called, it returns USB_STATUS_ERROR.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint that was provided via USBD_AddEP(). This data can be retrieved by a later call to USBD_Receive() / USBD_Read(). See also USBD_GetNumBytesInBuffer().

If a timeout occurs, the read transfer is not affected. Data sent from the host after the timeout is stored into the internal buffer of the endpoint and can be read by later calls to USBD_ReceivePoll().

If Timeout = 0, the function behaves like USBD_Receive().

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.5 USBD_ReadAsync()

Description

Reads data from the host asynchronously. The function does not wait for the data to be received. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_ReadAsync(unsigned                EPIndex,
                    USB_ASYNC_IO_CONTEXT * pContext,
                    int                    ShortRead);
```

Parameters

Parameter	Description
EPIndex	Handle to an OUT endpoint returned by USBD_AddEP().
pContext	Pointer to an I/O context containing parameters and pointer to the callback function.
ShortRead	• 0: The transfer is completed successfully after all bytes have been read. • 1: The transfer is completed successfully after one packet has been read. • 2: The transfer is completed successfully after a short packet has been read.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.6 USBD_Write()

Description

Writes data to the host. Depending on the Timeout parameter, the function may block until NumBytes have been written or a timeout occurs.

Prototype

```
int USBD_Write(      unsigned   EPIndex,
               const void     * pData,
                     unsigned   NumBytes,
                     I8         Send0PacketIfRequired,
                     int        ms);
```

Parameters

Parameter	Description
EPIndex	Handle to an IN endpoint returned by USBD_AddEP().
pData	Pointer to data that should be sent to the host.
NumBytes	Number of bytes to be written.
Send0PacketIfRequired	Specifies that a zero-length packet should be sent when the last data packet to the host is a multiple of MaxPacketSize.
ms	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0	Successfully starts an asynchronous write transfer, or a timeout has occurred and no data was written.
> 0 && < NumBytes	Number of bytes that have been written before a timeout occurred.
= NumBytes	The write transfer completed successfully.
< 0	An error occurred.

Additional information

This function also returns when the target is disconnected from the host or when a USB reset occurs.

The USB stack is able to queue a small number of asynchronous write transfers (Timeout = -1). If a write transfer is still in progress when this function is called and the USB stack cannot accept another write transfer request, the function returns USB_STATUS_EP_BUSY. A synchronous write transfer (Timeout ≥ 0) will always block until the transfer (including all pending transfers) finishes.

In order to synchronize, USBD_WaitForEndOfTransfer() needs to be called. Another synchronization method would be to periodically call USBD_GetNumBytesRemToWrite() in order to see how many bytes still need to be written (this method is preferred when a non-blocking solution is necessary).

In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 136).

The write operation can be canceled using USBD_CancelIO().

If pData = NULL and NumBytes = 0, a zero-length packet is sent to the host.

The content of the buffer pointed to by pData must not be changed until the transfer has been completed.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.7 USBD_WriteAsync()

Description

Sends data to the host asynchronously. The function does not wait for the data to be sent. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_WriteAsync(unsigned                EPIndex,
                     USB_ASYNC_IO_CONTEXT * pContext,
                     I8                     Send0PacketIfRequired);
```

Parameters

Parameter	Description
EPIndex	Handle to an IN endpoint returned by USBD_AddEP().
pContext	Pointer to an I/O context containing parameters and pointer to the callback function.
Send0PacketIfRequired	Specifies that a zero-length packet shall be sent when the last data packet is a multiple of MaxPacketSize.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.8 USBD_CancelIO()

Description

Cancel any read or write operation.

Prototype

```
void USBD_CancelIO(unsigned EPIndex);
```

Parameters

Parameter	Description
EPIndex	Handle to an endpoint returned by USBD_AddEP().

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.9 USBD_WaitForEndOfTransferEx()

Description

Waits until the current transfer on a particular EP completes. This function must be called from a task.

Prototype

```
int USBD_WaitForEndOfTransferEx(unsigned EPIndex,
                                unsigned Timeout,
                                int      AbortOnTimeout);
```

Parameters

Parameter	Description
EPIndex	Handle to the endpoint returned by USBD_AddEP().
Timeout	Timeout in milliseconds, 0 means infinite wait.
AbortOnTimeout	If a timeout occurs, then the current transfer is terminated if AbortOnTimeout ≠ 0. The current transfer is not affected in case of a timeout if AbortOnTimeout = 0. See also *Timeout handling* on page 136.

Return value

0 Transfer completed.
1 Timeout occurred.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.10 USBD_WaitForTXReady()

Description

Waits (blocking) until the TX queue can accept another data packet. This function is used in combination with a non-blocking call to USBD_Write(), it waits until a new asynchronous write data transfer will be accepted by the USB stack.

Prototype

```
int USBD_WaitForTXReady(unsigned EPIndex,
                        int      Timeout);
```

Parameters

Parameter	Description
EPIndex	Handle to an IN endpoint returned by USBD_AddEP().
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is negative, the function will return immediately.

Return value

= 0 A new asynchronous write data transfer will be accepted.
= 1 The write queue is full, a call to USBD_Write() would return USB_STATUS_EP_BUSY.
< 0 Error occurred.

Additional information

If Timeout is 0, the function never returns 1. If Timeout is -1, the function will not wait, but immediately return the current state.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.11 USBD_GetNumBytesInBuffer()

Description

Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer. This function does not start a read transfer.

Prototype

```
unsigned USBD_GetNumBytesInBuffer(unsigned EPIndex);
```

Parameters

Parameter	Description
EPIndex	Handle to an OUT endpoint returned by USBD_AddEP().

Return value

Number of bytes which have been stored in the internal buffer.

Additional information

The number of bytes returned by this function can be read using USBD_Read() or USBD_Receive() without blocking.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.12 USBD_GetNumBytesRemToRead()

Description

This function is to be used in combination with `USBD_ReadOverlapped()`. It returns the number of bytes which still have to be read during the transaction.

Prototype

```
unsigned USBD_GetNumBytesRemToRead(unsigned EPIndex);
```

Parameters

Parameter	Description
EPIndex	Handle to an OUT endpoint returned by `USBD_AddEP()`.

Return value

Number of bytes which still have to be read.

Additional information

Note that this function does not return the number of bytes that have been read, but the number of bytes which still have to be read. This function does not block.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.13 USBD_GetNumBytesRemToWrite()

Description

This function is to be used in combination with a non-blocking call to `USBD_Write()`. It returns the number of bytes which still have to be written during the transaction.

Prototype

```
unsigned USBD_GetNumBytesRemToWrite(unsigned EPIndex);
```

Parameters

Parameter	Description
EPIndex	Handle to an IN endpoint returned by `USBD_AddEP()`.

Return value

Number of bytes which still have to be written.

Additional information

Note that this function does not return the number of bytes that have been written, but the number of bytes which still have to be written. This function does not block.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.3.14 USBD_StallEP()

Description

Stalls an endpoint.

Prototype

```
void USBD_StallEP(unsigned EPIndex);
```

Parameters

Parameter	Description
EPIndex	Handle to the endpoint handle returned by USBD_AddEP().

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.4 USB Remote wakeup functions

Remote wakeup is a feature that allows a device to wake a host system from a USB suspend state.

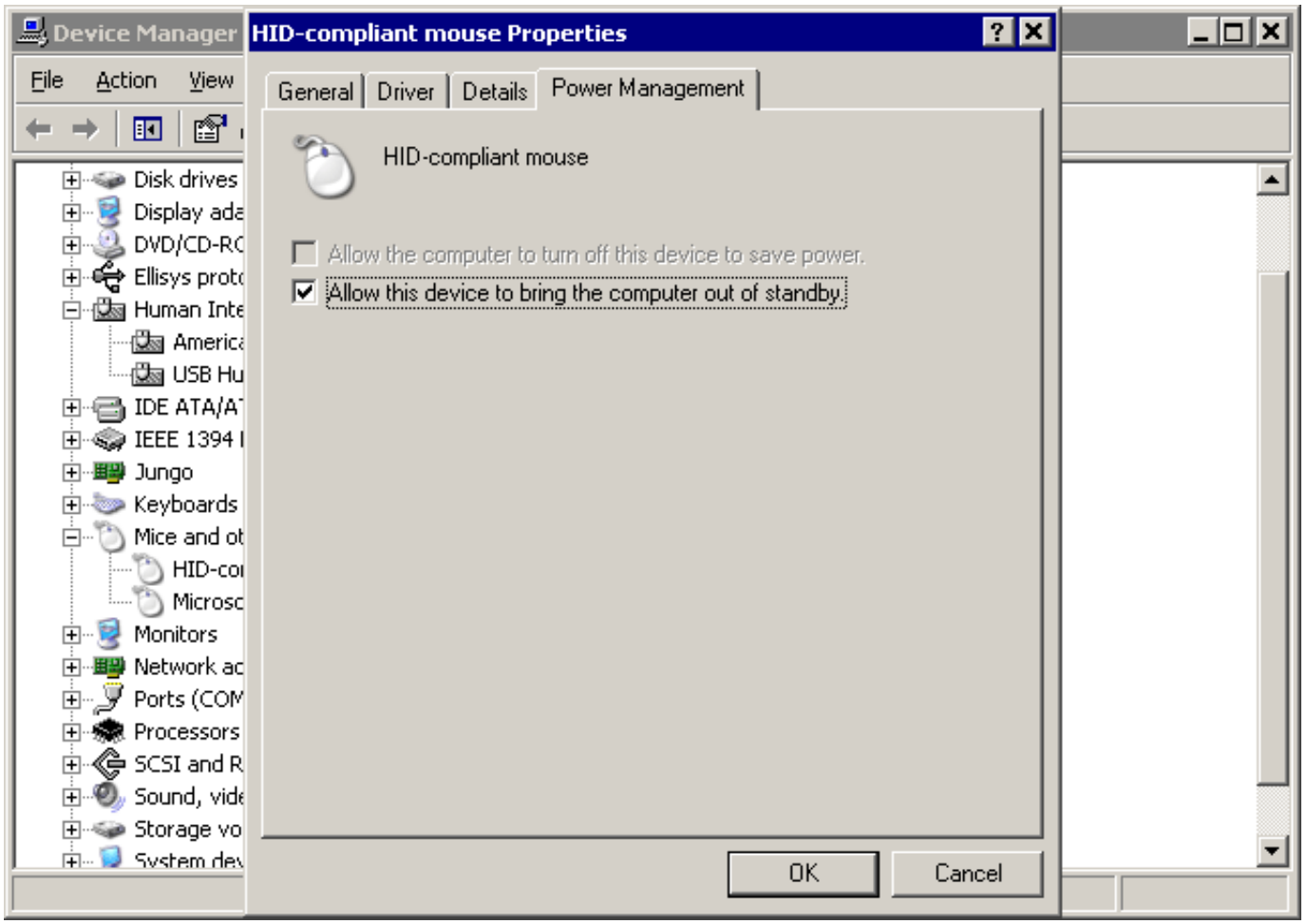
In order to do this, a special resume signal is sent over the USB data lines.

Additionally, the USB host controller and operating system have to be able to handle this signaling.

Windows OS

Currently, Windows OS only supports the wakeup feature on devices based on HID mouse/ keyboard, CDC Modem and RNDIS Ethernet class. Remote wakeup for MSD, generic bulk and CDC serial is not supported by Windows. Therefore a HID mouse class even as a dummy interface within your USB configuration is currently mandatory.

Windows must also be told that the device shall wake the PC from the suspend state. This is done by setting the option “Allow this device to bring the computer out of standby”.



macOS

macOS supports remote wakeup for all device classes.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.4.1 USBD_SetAllowRemoteWakeUp()

Description

Allows the device to publish that remote wake is available.

Prototype

```
void USBD_SetAllowRemoteWakeUp(U8 AllowRemoteWakeup);
```

Parameters

Parameter	Description
AllowRemoteWakeup	• 1 - Allows and publishes that remote wakeup is available. • 0 - Publish that remote wakeup is not available.

Additional information

This function must be called before the function `USBD_Start()` is called. This ensures that the Host is informed that USB remote wake up is available.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.4.2 USBD_DoRemoteWakeup()

Description

Performs a remote wakeup in order to wake up the host from the standby/suspend state. This will only work, when the USB device driver supports this. The function must be called only, if either:

A) The USB device is in suspend state: (`USBD_GetState()` & `USB_STAT_SUSPENDED`) ≠ 0 and remote wakeup is allowed by the host: (`USBD_GetDeviceState()` & `USB_DEVSTAT_REMOTE_WAKEUP_ALLOWED`) ≠ 0.

OR

B) The USB bus is in L1 state and remote wakeup is allowed by the host, see `USBD_SetOnLPMChange()`.

Prototype

```
void USBD_DoRemoteWakeup(void);
```

Additional information

This function cannot be called from an ISR context.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.5 Data structures

4.2.5.1 USB_ADD_EP_INFO

Description

Structure used by USBD_AddEPEx() when adding an endpoint.

Type definition

```
typedef struct _USB_ADD_EP_INFO {
  unsigned   MaxPacketSize;
  U16        Interval;
  U8         Flags;
  U8         InDir;
  U8         TransferType;
  U8         ISO_Type;
} USB_ADD_EP_INFO;
```

Structure members

Member	Description
MaxPacketSize	Maximum packet size for the endpoint.
Interval	Specifies the interval measured in units of 125us (microframes). This value should be zero for a bulk endpoint.
Flags	Specifies whether optional parameters are used. • 0x00 - Ignore optional parameters. • USB_ADD_EP_FLAG_USE_ISO_SYNC_TYPES - Use ISO_Type. If not set the endpoint will have the sync type USB_ISO_SYNC_TYPE_NONE.
InDir	Specifies the direction of the desired endpoint. • USB_DIR_IN • USB_DIR_OUT
TransferType	Specifies the transfer type of the endpoint. The following values are allowed: • USB_TRANSFER_TYPE_BULK • USB_TRANSFER_TYPE_ISO • USB_TRANSFER_TYPE_INT
ISO_Type	Allows to set the synchronization type for isochronous endpoints. The following types are supported: • USB_ISO_SYNC_TYPE_NONE (default) • USB_ISO_SYNC_TYPE_ASYNCHRONOUS • USB_ISO_SYNC_TYPE_ADAPTIVE • USB_ISO_SYNC_TYPE_SYNCHRONOUS

Additional information

The Interval parameter specifies the frequency in which the endpoint should be polled for information by the host. It must be specified in units of 125 us. Depending on the actual speed of the device during enumeration, the USB stack converts the interval to the correct value required for the endpoint descriptor according to the USB specification (into milliseconds for low/full-speed, into 125 us for high-speed). For endpoints of type USB_TRANSFER_TYPE_BULK the value is ignored and should be set to 0.

The actual maximum packet size for bulk endpoints may be smaller than given in the 'MaxPacketSize' field to meet the requirements of the actual USB speed.

For SuperSpeed bulk endpoints, MaxPacketSize can be N * 1024, where N = 1…16. Values of N > 1 enables the usage of burst transfers.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.5.2 USB_SETUP_PACKET

Description

Structure containing a USB setup packet received from the host.

Type definition

```
typedef struct {
  U8   bmRequestType;
  U8   bRequest;
  U8   wValueLow;
  U8   wValueHigh;
  U8   wIndexLow;
  U8   wIndexHigh;
  U8   wLengthLow;
  U8   wLengthHigh;
} USB_SETUP_PACKET;
```

Structure members

Member	Description
bmRequestType	Setup request type.
bRequest	Setup request number.
wValueLow	Low byte of the value field.
wValueHigh	High byte of the value field.
wIndexLow	Low byte of the index field.
wIndexHigh	High byte of the index field.
wLengthLow	Low byte of the length field.
wLengthHigh	High byte of the length field.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.5.3 SEGGER_CACHE_CONFIG

Description

Used to pass cache configuration and callback function pointers to the stack.

Prototype

```
typedef struct {
  unsigned int CacheLineSize;
  void (*pfDMB)       (void);
  void (*pfClean)     (void *p, unsigned long NumBytes);
  void (*pfInvalidate)(void *p, unsigned long NumBytes);
} SEGGER_CACHE_CONFIG;
```

Member	Description
CacheLineSize	Cache line size of the CPU in bytes. Most systems use a 32-byte cache line size.
pfDMB	Unused.
pfClean	Pointer to a callback function that executes a clean operation on cached memory. The parameter 'p' is always cache aligned. 'NumBytes' must be rounded up by the function to the next multiple of the cache line size, if necessary.
pfInvalidate	Pointer to a callback function that executes an invalidate operation on cached memory. The parameter 'p' is always cache aligned. 'NumBytes' must be rounded up by the function to the next multiple of the cache line size, if necessary.

Additional information

For further information about how this structure is used, please refer to *USBD_SetCacheConfig* on page 97.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.5.4 USB_CHECK_ADDRESS_FUNC

Description

Checks if an address can be used for DMA transfers. The function must return 0, if DMA access is allowed for the given address, 1 otherwise.

Type definition

```
typedef int USB_CHECK_ADDRESS_FUNC(const void * pMem);
```

Parameters

Parameter	Description
pMem	Pointer to the memory.

Return value

= 0 Memory can be used for DMA access.
≠ 0 DMA access not allowed for the given address.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.5.5 USB_ASYNC_IO_CONTEXT

Description

Contains information for asynchronous transfers.

Type definition

```
typedef struct _USB_ASYNC_IO_CONTEXT {
  unsigned                    NumBytesToTransfer;
  void                      * pData;
  USB_ASYNC_CALLBACK_FUNC   * pfOnComplete;
  void                      * pContext;
  int                         Status;
  unsigned                    NumBytesTransferred;
} USB_ASYNC_IO_CONTEXT;
```

Structure members

Member	Description
NumBytesToTransfer	Number of bytes to transfer. Must be set by the application.
pData	Pointer to the buffer for read operations, pointer to the data for write operations. Must be set by the application.
pfOnComplete	Pointer to the function called on completion of the transfer. Must be set by the application.
pContext	Pointer to a user context. Can be arbitrarily used by the application.
Status	Result status of the asynchronous transfer. Set by the USB stack before calling pfOnComplete.
NumBytesTransferred	Number of bytes transferred. Set by the USB stack before calling pfOnComplete.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.5.6 USB_WEBUSB_INFO

Description

Information that may be provided by the application for WebUSB capable USB devices. Can be set via the function `USBD_SetWebUSBInfo()` before the USB stack is started using `USBD_Start()`. Is used during enumeration of the device by the host.

Type definition

```
typedef struct {
  U8            VendorCode;
  U8            DescIndex;
  U8            URLPrefix;
  const char  * sURL;
} USB_WEBUSB_INFO;
```

Structure members

Member	Description
VendorCode	Vendor code used for the setup request.
DescIndex	Descriptor index of the descriptor containing the URL of the landing page.
URLPrefix	Prefix of the URL: 0 = “http://”, 1 = “https://”, 255 = none.
sURL	URL of the landing page. UTF-8 string.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.6 Function Types

4.2.6.1 USB_ON_CLASS_REQUEST

Description

Type of callback set in `USBD_SetClassRequestHook()` or `USBD_SetVendorRequestHook()`. This function is called when a setup class request is sent from the host to the specified interface index.

Type definition

```
typedef int USB_ON_CLASS_REQUEST(const USB_SETUP_PACKET * pSetupPacket);
```

Parameters

Parameter	Description
`pSetupPacket`	Pointer to the setup packet received from the host.

Return value

If the function has processed the setup packet, it must acknowledge the packet by either sending a response packet using `USBD_WriteEP0FromISR()` or an empty packet with `USBD_WriteEP0FromISR(NULL, 0, 0)` and must return 0.

If the function can't process the packet, it must return 1. In this case the USB stack tries to process the packet and will send a STALL if that fails.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.6.2 USB_ON_SETUP

Description

Type of callback set in `USBD_SetOnSetup()`. This function is called when a setup request was sent from the host.

Type definition

```
typedef int USB_ON_SETUP(const USB_SETUP_PACKET * pSetupPacket);
```

Parameters

Parameter	Description
`pSetupPacket`	Pointer to the setup packet received from the host.

Return value

If the function has processed the setup packet, it must acknowledge the packet by either sending a response packet using `USBD_WriteEP0FromISR()` or an empty packet with `USBD_WriteEP0FromISR(NULL, 0, 0)` and must return 0.

If the function can't process the packet, it must return 1. In this case the USB stacks tries to process the packet and will send a STALL if that fails.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.6.3 USB_GET_STRING_FUNC

Description

Type of callback set in `USBD_SetGetStringHook()`. This function is called when a string descriptor is requested from the host.

Type definition

```
typedef const char * USB_GET_STRING_FUNC(int Index);
```

Parameters

Parameter	Description
`Index`	`Index` of the requested string.

Return value

If the function is able to provide a string for the given index, it should return a pointer to an ASCII string. Otherwise it should return a `NULL` pointer.

 © 2010-2026 SEGGER Microcontroller GmbH

4.2.6.4 USB_ON_LPM_CHANGE

Description

Type of callback set in USBD_SetOnLPMChange(). This function is called when a LPM transition on the USB lines (L0 <-> L1) is detected.

Type definition

```
typedef void USB_ON_LPM_CHANGE(int      State,
                               unsigned BESL);
```

Parameters

Parameter	Description
State	• -1 - Transition to L0. • 0 - Transition to L1. Remote wakeup not allowed. • 1 - Transition to L1. Remote wakeup allowed.
BESL	BESL value (Best Effort Service Latency) in range 0…15 reported by the host when requesting a transition to L1 state. Values of 0,1,…,14,15 specify a BESL of 125us,150us, …,9000us,10000us respectively, see “Errata for USB 2.0 ECN: Link Power Management (LPM) - 7/2007” from usb.org for an explanation of these values.

 © 2010-2026 SEGGER Microcontroller GmbH

4.3 Timeout handling

Many API functions have a timeout parameter that causes the functions to return if the desired transaction cannot be finished within the given time. Hardware USB controllers usually do not have a mechanism for timeouts. Therefore, the USB stack has to handle timeouts as follows:

- Start a transaction.
- Wait for the transaction to complete or the timeout to expire.
- If the timeout has expired: Abort the current transaction.

Aborting a transaction is always a critical operation. The USB software is informed by the hardware only if a transaction has been completed. The software usually does not know if a data transfer on the USB lines is still in progress. So if the USB stack decides to abort a transaction, this transaction may already be in progress at that time. In this case the abort of the transfer may cause the data currently transferred to be discarded without any notice to the software. Although the data packet was successfully transferred on the USB bus and acknowledged by the host, the data is lost from the target application's viewpoint.

Because this is usually not the behavior intended by the application, timeouts should be used to handle fatal errors only. Timeouts should not be used to repeatedly poll for data.

Bad example

NOT RECOMMENDED

```
for (;;) {
  // Try to read some data with 5 ms timeout
  NumBytesRead = USBD_Receive(EP, Buffer, 100, 5);
  if (NumBytesRead < 0) {
    <handle error>
    break;
  }
  if (NumBytesRead > 0) {
    <process the data>
    continue;
  }
  // NumBytesRead is 0 here, that means a timeout has occurred
  <execute other tasks>
  // Repeat the loop and retry to read data
}
```

In this example, data packets may be lost if they arrive exactly when the 5 ms timeout expires.

There are several options to avoid this problem:

- Using non-blocking API functions, like `USBD_Receive()` where the Timeout value = -1, e.g. `USBD_Receive(EP, Buffer, 100, -1)`.
- Using asynchronous API functions, like `USBD_ReadOverlapped()`, `USBD_ReadAsync()`.
- Using blocking API functions with a timeout that don't abort the transaction. They usually have a "Poll" in their name. The above example works well when using the function `USBD_ReceivePoll()` instead of `USBD_Receive()`.

The same applies when writing data to the host.

 © 2010-2026 SEGGER Microcontroller GmbH

4.4 USB disconnect

If a self-powered USB device is disconnected from the host, the following happens:

1. USB suspend is detected. The global state (see `USBD_GetState()`) changes to `USB_STAT_ATTACHED|USB_STAT_READY|USB_STAT_ADDRESSED|USB_STAT_CONFIGURED|USB_STAT_SUSPENDED`.
2. All active and pending USB transfers are terminated. USB read and write API functions return with an error condition.
3. After some delay, the global state changes to `USB_STAT_ATTACHED`.

Some USB controllers are not able to distinguish between *USB suspend* state and USB disconnect. In these cases, step 3 is not applicable; instead, the USB state remains in `USB_STAT_SUSPENDED`. In order to detect a disconnect, the VBUS state has to be considered: If the stack turns into suspend state while VBUS is still present, the host has issued a suspend and a later resume (or remote wake-up) may be possible. If the stack signals a suspend event and VBUS is off, then the host was disconnected and no resume (or remote wake-up) is possible.

Note that this section is not applicable for bus-powered devices, as they will power off on a USB disconnect.

 © 2010-2026 SEGGER Microcontroller GmbH

4.5 Low power mode

emUSB-Device does not directly support low power modes of the device running USB, because it is very specific to the actual hardware and requirements of the application and there may be several different low power states. Low power mode may include:

- Shutting down peripherals (including the USB controller and/or the PHY)
- Shutting down PLLs
- Lowering the system clock
- CPU sleep modes

The device is usually put into low power state only if there is no USB connection to the host. Since the host supplies the device via (5 Volt) VBUS, there is no need for power saving. Without a USB host connection, the device may run from a battery which requires low power consumption.

The application is responsible for determining when low power state should be entered or exited. In most cases, it depends on VBUS: Enter low power mode while no VBUS is present. There is no general way to detect VBUS with the USB controller, especially if the USB controller is shut down. Therefore, VBUS detection must be managed by the application.

To enter low power mode, the application should:

- Call `USBD_Stop()`
- Enter low power mode

To leave low power mode, the application should:

- Resume from low power mode
- Call `USBD_Start()`

Alternately the USB stack may be re-initialized completely.

To enter low power mode:

- Call `USBD_DeInit()`
- Enter low power mode

To leave low power mode:

- Resume from low power mode
- Call `USBD_Init()`
- Call all necessary USB configuration functions (like `USBD_SetDeviceInfo()`, `USBD_<class>_Add()`, …)
- Call `USBD_Start()`

The second approach is necessary, for example, if the configuration which was done in the `USBD_X_Config()` function should be executed after resuming from low power mode, or if the memory used by the USB stack was shut down in low power mode and has lost its contents.

4.5.1 USB suspend

If the application wants to respond to a USB suspend from the host while the device stays connected to the host, it may simply monitor the status bit `USB_STAT_SUSPENDED` returned by the function `USBD_GetState()`. The USB stack must remain active to get correct states from `USBD_GetState()`.

Please notice that on some devices the USB controller is not able to distinguish between suspend state and disconnect, see *USB disconnect*.

The device may be put into low power mode during suspend. If the USB controller is affected by the low power mode (for example if the USB controller register settings are not retained), then the application has to save and restore the USB controller’s state before entering / after leaving low power state.

 © 2010-2026 SEGGER Microcontroller GmbH

Application design to handle suspend state

When a USB device is put into suspend state by the host, then all active and pending USB transfers are terminated. If for example the application is waiting for data from the host using:

```
  NumBytesReceived = USBD_CDC_Receive(hInst, Buff, sizeof(Buff), 0);
```

then the function `USBD_CDC_Receive()` will return with a negative value in case of a device suspend. Further calls to `USBD_CDC_Receive()` will also return immediately until the device is resumed. The application on the device has to handle this case properly. For example:

```
  ...
  USBD_Start();
  //
  // Wait for successful enumeration
  //
restart:
  while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
         != USB_STAT_CONFIGURED) {
    USB_OS_Delay(50);
  }
  //
  // Main application loop
  //
  for (;;) {
    //
    // Read data from host
    //
    NumBytesReceived = USBD_CDC_Receive(hInst, Buff, sizeof(Buff), 0);
    if (NumBytesReceived < 0) {
      if ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
           != USB_STAT_CONFIGURED) {
        //
        // device was disconnected or is in suspend state
        //
        goto restart;
      }
      //
      // error handling...
      //
      ...
    }
    //
    // Process data from host
    //
    ...
  }
```

4.5.2 Link Power Management (LPM)

To enable LPM, the application has to call `USBD_UseV210()` within the configuration function `USBD_X_Config()`. This sets the USB version of the device to 2.10. The host will then request the LPM capabilities from the device (contained in the BOS descriptor) during enumeration. The USB stack will offer LPM support only if the driver and the USB controller support it.

For SuperSpeed devices LPM is enabled by default.

Please notice that common USB hosts (Windows/Linux/macOS) use LPM for full- and high-speed devices only in special situations. If the host contains controller hardware other than an XHCI type controller, then LPM is not used. Also, if the device is not directly connected to that USB controller, but instead via a hub, then LPM is not used.

See also:

 © 2010-2026 SEGGER Microcontroller GmbH

- `USBD_SetBESLValues()`
- `USBD_SetOnLPMChange()`
- `USBD_SetLPMResponse()`

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 5

Bulk communication

This chapter describes how to get emUSB-Device-Bulk up and running.



 © 2010-2026 SEGGER Microcontroller GmbH

5.1 Generic bulk stack

The generic bulk stack is located in the directory *USB*. All C files in the directory should be included in the project (compiled and linked as part of your project). The files in this directory are maintained by SEGGER and should not require any modification. All files requiring modifications have been placed in other directories.

 © 2010-2026 SEGGER Microcontroller GmbH

5.2 Requirements for the Host (PC)

In order to communicate with a target (client) running emUSB-Device, the operating system running on the host must recognize the device connected to it.

5.2.1 Windows

Microsoft's Windows operating systems contain a generic driver called WinUSB.sys that is used to handle all communication to an emUSB-Device running a BULK interface. If an emUSB bulk device is connected to a Windows 8, 8.1, 10 or 11 PC for the first time, Windows will install the WinUSB driver automatically. For Windows versions less than Windows 8, Microsoft provides a driver for Windows Vista and Windows 7, but this needs to be installed manually. A driver installation tool including the mentioned driver is available in the `Windows\USB\Bulk\WinUSBInstall`. In order to get emUSB BULK running with the WinUSB driver, the following must be considered:

- The function `USBD_BULK_SetMSDescInfo()` must be called in the target application.
- The Product IDs 1234 and 1121 must not be used.

5.2.2 Linux

Linux can handle emUSB BULK devices out of the box.

By default, a USB device can only be accessed by a process that is running with "root" rights. In order to use the USB bulk device from normal user programs, an `udev` rule has to be configured for the device (refer to the Linux `udev` documentation). The emUSB-Device release contains a sample configuration file `99-emUSBD.rules`, which may be modified and copied to /etc/udev/rules.d on the host machine.

5.2.3 macOS

macOS can handle emUSB BULK devices out of the box.

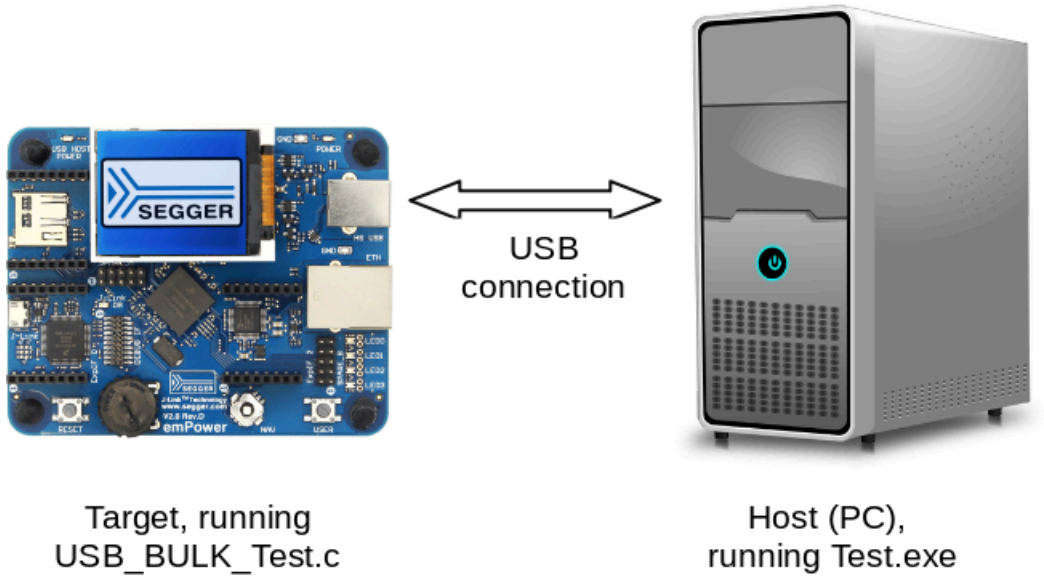
 © 2010-2026 SEGGER Microcontroller GmbH

5.3 Example application

Example applications for both the target (client) and the PC (host) are supplied. These can be used for testing the correct installation and proper function of the device running emUSB-Device.

The host sample applications can be used for Windows, Linux and macOS. Precompiled executables for Windows can be found in the subfolder `Windows/USB/Bulk/SampleApplication/Exe`.

The application `USB_BULK_Test.c` is a modified echo server; the application receives data, modifies the first byte and sends it back to the host. It also contains the functionality to measure USB transfer speed.



To use this application, make sure to use the corresponding example files both on the host side and on the target side. The example applications on the PC host are named in the same way, just without the prefix `USB_BULK_`.

The example applications for the target side are supplied in source code in the `Application` directory.

For information on how to compile the host examples (especially for Linux and macOS) refer to *Compiling the PC example application* on page 145.

The start application will of course later on be replaced by the real application program. For the purpose of getting emUSB-Device up and running as well as doing an initial test, the start application should not be modified.

5.3.1 Running the example applications

To test the emUSB-Device-Bulk component, build and download the application of choice for the target-side.

To run one of the example applications, simply start the executable, for example by double-clicking it.

If a connection can be established, it exchanges data with the target, testing the USB connection.

Example output of `Test.exe`:

 © 2010-2026 SEGGER Microcontroller GmbH

```
rxvt
$ Exe/Test

Found 1 device
Found the following device 0:
  Vendor Name : Vendor
  Product Name: Bulk test
  Serial no.  : 13245678
To which device do you want to connect?
Please type in device number (e.g. '0' for the first device, q/a for abort):0
Echo test
Operation successful!

Read speed test
................................................................................................
................................................................................................
..........................................................
Performance: 6145 ms for 256 MB
          =  42659 kB / second

Write speed test
................................................................................................
................................................................................................
..........................................................
Performance: 6144 ms for 256 MB
          =  42666 kB / second

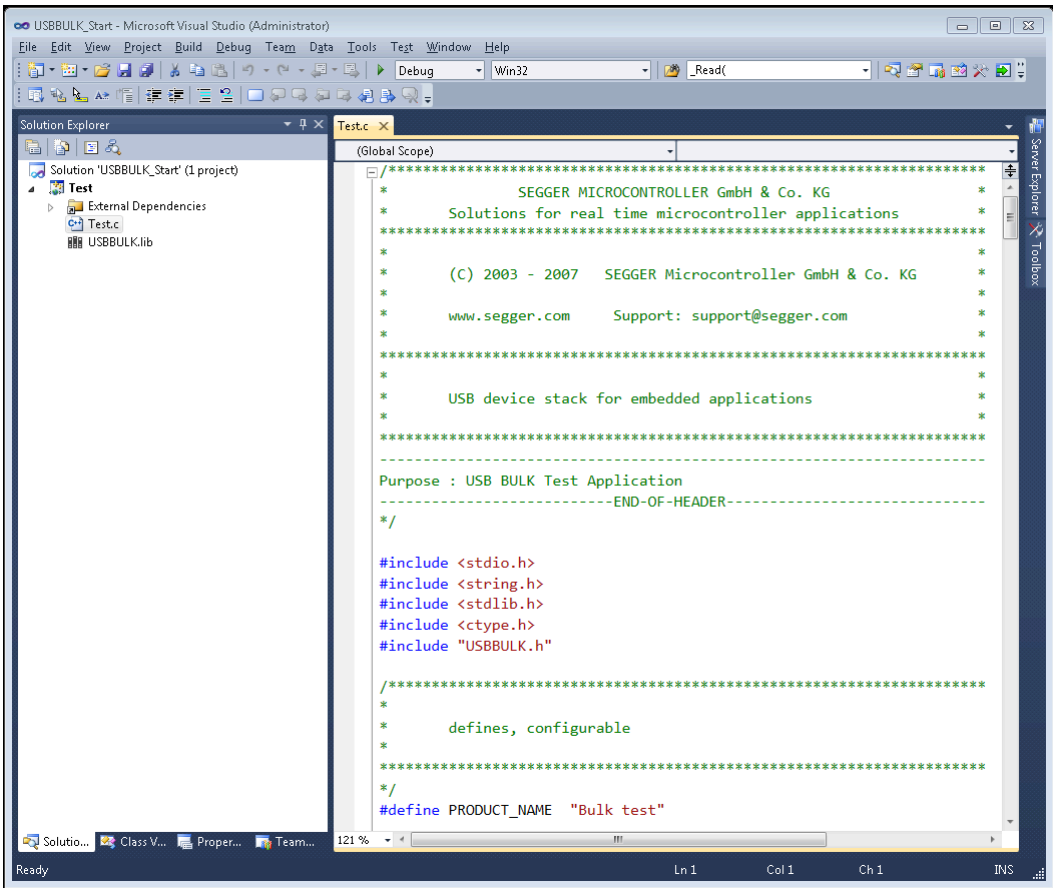
Echo test
Operation successful!

Communication with USB BULK device was successful!
Press enter to exit.
```

5.3.2 Compiling the PC example application

5.3.2.1 Windows

For compiling the example application you need Visual C++ 2010 (or later).



The source code of the sample application is located in the subfolder `Windows/USB/BULK/SampleApplication/Src`. Open the file `USBBULK_Start.sln` and compile the source.

 © 2010-2026 SEGGER Microcontroller GmbH

5.3.2.2 Linux

The subfolder `Windows/USB/Bulk/SampleApplication` contains a Makefile for Linux. Change to this folder and execute “make”.

5.3.2.3 macOS

The subfolder `Windows/USB/Bulk/SampleApplication` contains a Makefile for macOS. Change to this folder and execute “make -f `Makefile_MacOSX`”.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4 Target API

This chapter describes the functions that can be used with the target system.

General information

To communicate with the host, the sample application project includes USB-specific header and source files (`USB.h`, `USB_Main.c`, `USB_Setup.c`, `USB_Bulk.c`, `USB_Bulk.h`). These files contain API functions to communicate with the USB host through the emUSB-Device driver.

Purpose of the USB Device API functions

To have an easy startup when writing an application on the device side, these API functions have a simple interface and handle all operations that need to be done to communicate with the host's kernel.

Therefore, all operations that need to be performed to write to or read from the emUSB-Device are handled internally by the provided API functions.

5.4.1 Target interface function list

Routine	Explanation
USB-Bulk functions	
`USBD_BULK_Add()`	Adds interface for USB-Bulk communication to emUSB-Device.
`USBD_BULK_Add_Ex()`	Adds interface for USB-Bulk communication to emUSB-Device.
`USBD_BULK_AddAlternateInterface()`	Adds an alternative interface for USB-Bulk interface.
`USBD_BULK_SetMSDescInfo()`	Enables use of Microsoft OS Descriptors.
`USBD_BULK_CancelRead()`	Cancels any non-blocking/blocking read operation that is pending.
`USBD_BULK_CancelWrite()`	Cancels any non-blocking/blocking write operation that is pending.
`USBD_BULK_GetNumBytesInBuffer()`	Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer.
`USBD_BULK_GetNumBytesRemToRead()`	Get the number of remaining bytes to read by an active read operation.
`USBD_BULK_GetNumBytesRemToWrite()`	After starting a non-blocking write operation this function can be used to periodically check how many bytes still have to be written.
`USBD_BULK_Read()`	Reads data from the host with a given timeout.
`USBD_BULK_ReadAsync()`	Reads data from the host asynchronously.
`USBD_BULK_ReadOverlapped()`	Reads data from the host asynchronously.
`USBD_BULK_ReadRecord()`	Reads data from host.
`USBD_BULK_Receive()`	Reads data from the host.
`USBD_BULK_ReceivePoll()`	Reads data from the host.
`USBD_BULK_SetContinuousReadMode()`	Enables continuous read mode for the RX endpoint.
`USBD_BULK_SetOnSetupRequest()`	Sets a callback function that is called when any setup request is sent from the host.

 © 2010-2026 SEGGER Microcontroller GmbH

Routine	Explanation
USBD_BULK_SetOnRXEvent()	Sets a callback function for the OUT endpoint that will be called on every RX event for that endpoint.
USBD_BULK_SetOnTXEvent()	Sets a callback function for the IN endpoint that will be called on every TX event for that endpoint.
USBD_BULK_TxIsPending()	Checks whether the TX (IN endpoint) is currently pending.
USBD_BULK_WaitForRX()	Waits (blocking) until the triggered USBD_BULK_ReadOverlapped() has received the desired data.
USBD_BULK_PollForRX()	Waits (blocking) until the triggered USBD_BULK_ReadOverlapped() has received the desired data.
USBD_BULK_WaitForTX()	Waits (blocking) until a pending asynchronous USBD_BULK_Write() (Timeout parameter = -1) has sent the desired data.
USBD_BULK_PollForTX()	Waits (blocking) until a pending asynchronous USBD_BULK_Write() (Timeout parameter = -1) has sent the desired data.
USBD_BULK_WaitForTXReady()	Waits (blocking) until the TX queue can accept another data packet.
USBD_BULK_Write()	Sends data to the USB host.
USBD_BULK_WriteAsync()	Sends data to the host asynchronously.
USBD_BULK_WriteEx()	Send data to the USB host with NULL packet control.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2 USB-Bulk functions

5.4.2.1 USBD_BULK_Add()

Description

Adds interface for USB-Bulk communication to emUSB-Device.

Prototype

```
USB_BULK_HANDLE USBD_BULK_Add(const USB_BULK_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to USB_BULK_INIT_DATA structure.

Return value

Handle to a valid BULK instance. The handle of the first BULK instance is always 0.

Example

Example excerpt from BULK_Echo1.c:

```
static void _AddBULK(void) {
  static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
  USB_BULK_INIT_DATA    Init;
  Init.EPIn  = USBD_AddEP(USB_DIR_IN,
                          USB_TRANSFER_TYPE_BULK,
                          USB_HS_BULK_MAX_PACKET_SIZE,
                          NULL,
                          0);
  Init.EPOut = USBD_AddEP(USB_DIR_OUT,
                          USB_TRANSFER_TYPE_BULK,
                          USB_HS_BULK_MAX_PACKET_SIZE,
                          _abOutBuffer,
                          USB_HS_BULK_MAX_PACKET_SIZE);
  USBD_BULK_Add(&Init);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.2 USBD_BULK_Add_Ex()

Description

Adds interface for USB-Bulk communication to emUSB-Device.

Prototype

```
USB_BULK_HANDLE USBD_BULK_Add_Ex(const USB_BULK_INIT_DATA_EX * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to USB_BULK_INIT_DATA_EX structure.

Return value

Handle to a valid BULK instance. The handle of the first BULK instance is always 0.

Example

```
static void _AddBULK(void) {
  static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
  USB_BULK_INIT_DATA_EX  Init;
  Init.Flags = 0;
  Init.EPIn  = USBD_AddEP(USB_DIR_IN,
                          USB_TRANSFER_TYPE_BULK,
                          USB_HS_BULK_MAX_PACKET_SIZE,
                          NULL,
                          0);
  Init.EPOut = USBD_AddEP(USB_DIR_OUT,
                          USB_TRANSFER_TYPE_BULK,
                          USB_HS_BULK_MAX_PACKET_SIZE,
                          _abOutBuffer,
                          USB_HS_BULK_MAX_PACKET_SIZE);
  Init.pInterfaceName = "BULK Interface";
  USBD_BULK_Add_Ex(&Init);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.3 USBD_BULK_AddAlternateInterface()

Description

Adds an alternative interface for USB-Bulk interface.

Prototype

```
void USBD_BULK_AddAlternateInterface(      USB_BULK_HANDLE             hInst,
                                     const USB_BULK_INIT_DATA_EX     * pInitData,
                                           USB_ON_USER_SET_INTERFACE * pfOnUser);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add-d().
pInitData	Pointer to USB_BULK_INIT_DATA_EX structure.
pfOnUser	Callback function that is called, when the host changes the interface.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.4 USBD_BULK_SetMSDescInfo()

Description

Enables use of Microsoft OS Descriptors. A USB bulk device providing these descriptors is detected by Windows to be handled by the generic WinUSB driver. For such devices no other driver needs to be installed.

Prototype

```
void USBD_BULK_SetMSDescInfo(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Additional information

This function must be called after the call to the function USBD_BULK_Add() and before USBD_Start().

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.5 USBD_BULK_CancelRead()

Description

Cancels any non-blocking/blocking read operation that is pending.

Prototype

```
void USBD_BULK_CancelRead(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Additional information

This function shall be called when a pending asynchronous read operation should be canceled. The function can be called from any task. In case of canceling a blocking operation, this function must be called from another task.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.6 USBD_BULK_CancelWrite()

Description

Cancels any non-blocking/blocking write operation that is pending.

Prototype

```
void USBD_BULK_CancelWrite(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Additional information

This function shall be called when a pending asynchronous write operation should be canceled. The function can be called from any task. In case of canceling a blocking operation, this function must be called from another task.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.7 USBD_BULK_GetNumBytesInBuffer()

Description

Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer. This function does not start a read transfer.

Prototype

```
unsigned USBD_BULK_GetNumBytesInBuffer(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Return value

Number of bytes that are available in the internal BULK-OUT endpoint buffer.

Additional information

If the host is sending more data than your target application has requested, the remaining data will be stored in an internal buffer. This function shows how many bytes are available in this buffer.

The number of bytes returned by this function can be read using USBD_BULK_Read() without blocking.

Example

Your host application sends 50 bytes. Your target application only requests to receive 1 byte. In this case, the target application will get 1 byte and the remaining 49 bytes are stored in an internal buffer. When your target application now calls USBD_BULK_GetNumBytesInBuffer(), it will return the number of bytes that are available in the internal buffer (49).

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.8 USBD_BULK_GetNumBytesRemToRead()

Description

Get the number of remaining bytes to read by an active read operation. This function is to be used in combination with `USBD_BULK_ReadOverlapped()`. After starting the read operation this function can be used to periodically check how many bytes still have to be read.

Prototype

```
unsigned USBD_BULK_GetNumBytesRemToRead(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
`hInst`	Handle to a valid BULK instance, returned by `USBD_BULK_Add()`.

Return value

≥ 0 Number of bytes which have not yet been read.
< 0 Error occurred.

Additional information

Alternatively the blocking function `USBD_BULK_WaitForRX()` can be used.

Example

```
NumBytesReceived = USBD_BULK_ReadOverlapped(hInst, &ac[0], 50);
if (NumBytesReceived < 0) {
  <.. error handling..>
}
if (NumBytesReceived > 0) {
  // Already had some data in the internal buffer.
  // The first 'NumBytesReceived' bytes may be processed here.
  <...>
} else {
  // Wait until we get all 50 bytes
  while (USBD_BULK_GetNumBytesRemToRead(hInst) > 0) {
    USB_OS_Delay(50);
  }
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.9 USBD_BULK_GetNumBytesRemToWrite()

Description

After starting a non-blocking write operation this function can be used to periodically check how many bytes still have to be written.

Prototype

```
unsigned USBD_BULK_GetNumBytesRemToWrite(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Return value

Number of bytes which have not yet been written.

Additional information

Alternatively the blocking function USBD_BULK_WaitForTX() can be used.

Example

```
r = USBD_BULK_Write(hInst, &ac[0], TRANSFER_SIZE, -1);
if (r < 0) {
  <.. error handling..>
}
// NumBytesToWrite shows how many bytes still have to be written.
while (USBD_BULK_GetNumBytesRemToWrite(hInst) > 0) {
  USB_OS_Delay(50);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.10 USBD_BULK_Read()

Description

Reads data from the host with a given timeout.

Prototype

```
int USBD_BULK_Read(USB_BULK_HANDLE   hInst,
                   void            * pData,
                   unsigned          NumBytes,
                   unsigned          Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
Timeout	Timeout in milliseconds, 0 means infinite.

Return value

= NumBytes	Requested data was successfully read within the given timeout.
≥ 0 && < NumBytes	Timeout has occurred. Number of bytes that have been read within the given timeout.
< 0	Error occurred.

Additional information

This function blocks a task until all data have been read or a timeout expires. This function also returns when the device is disconnected from host or when a USB reset occurs.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via USBD_AddEP(). This data can be retrieved by a later call to USBD_BULK_Receive() / USBD_BULK_Read(). See also USBD_BULK_GetNumBytesInBuffer().

In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

If a read transfer was still pending while the function is called, it returns USB_STATUS_EP_BUSY.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.11 USBD_BULK_ReadAsync()

Description

Reads data from the host asynchronously. The function does not wait for the data to be received. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_BULK_ReadAsync(USB_BULK_HANDLE         hInst,
                         USB_ASYNC_IO_CONTEXT * pContext,
                         int                    ShortRead);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pContext	Pointer to a structure of type USB_ASYNC_IO_CONTEXT containing parameters and a pointer to the callback function.
ShortRead	• 0: The transfer is completed successfully after all bytes have been read. • 1: The transfer is completed successfully after one packet has been read. • 2: The transfer is completed successfully after a short packet has been read.

Example

```
static void _AsyncCb(USB_ASYNC_IO_CONTEXT * pIOContext) {
  U8  *p;

  p = (U8 *)pIOContext->pContext;
  *p = 1;
}

<...>

USB_ASYNC_IO_CONTEXT IOContext;
U8                   AsyncComplete;

IOContext.NumBytesToTransfer = 5000;
IOContext.pData              = pBuff;
IOContext.pfOnComplete       = _AsyncCb;
IOContext.pContext           = (void *)&AsyncComplete;
AsyncComplete = 0;
USBD_BULK_ReadAsync(hInst, &IOContext, 0);
while (AsyncComplete == 0) {
  <.. Do other work. ..>
}
// Transaction is complete.
if (IOContext.Status < 0 || IOContext.NumBytesTransferred != 5000) {
  <.. error handling ..>
} else {
  <.. Process the data ..>
}
<...>
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.12 USBD_BULK_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

```
int USBD_BULK_ReadOverlapped(USB_BULK_HANDLE   hInst,
                             void            * pData,
                             unsigned          NumBytes);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Return value

> 0 Number of bytes that have been read from the internal buffer (success).
= 0 No data was found in the internal buffer, read transfer started (success).
< 0 Error occurred.

Additional information

This function will not block the calling task. The read transfer will be initiated and the function returns immediately. In order to synchronize, USBD_BULK_WaitForRX() needs to be called. Alternatively the function USBD_BULK_GetNumBytesRemToRead() can be called periodically to check whether all bytes have been read or not. The read operation can be canceled using USBD_BULK_CancelRead(). The buffer pointed to by pData must be valid until the read operation is terminated.

If a read transfer was still pending while the function is called, it returns USB_STATUS_EP_BUSY.

Example

See *USBD_BULK_GetNumBytesRemToRead* on page 156.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.13 USBD_BULK_Receive()

Description

Reads data from the host. The function blocks until any data has been received or a timeout occurs (if `Timeout` ≥ 0). In contrast to `USBD_BULK_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received. In case of a timeout, the read transfer is aborted (see *`Timeout` handling* on page 136).

Prototype

```
int USBD_BULK_Receive(USB_BULK_HANDLE   hInst,
                      void            * pData,
                      unsigned          NumBytes,
                      int               Timeout);
```

Parameters

Parameter	Description
`hInst`	Handle to a valid BULK instance, returned by `USBD_BULK_Add()`.
`pData`	Pointer to a buffer where the received data will be stored.
`NumBytes`	Maximum number of bytes to read.
`Timeout`	`Timeout` in milliseconds. 0 means infinite. If `Timeout` is -1, the function never blocks and only reads data from the internal endpoint buffer.

Return value

> 0	Number of bytes that have been read.
= 0	A timeout occurred (if `Timeout` > 0), zero packet received (not every controller supports this!), no data in buffer (if `Timeout` < 0) or the target was disconnected during the function call and no data was read so far.
< 0	Error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USBD_BULK_Receive()` will return as much data as is currently available -- up to the size of the buffer specified. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return `USB_STATUS_ERROR`.

If a read transfer was pending while the function is called, it returns `USB_STATUS_EP_BUSY`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBD_AddEP()`. This data can be retrieved by a later call to `USBD_BULK_Receive()` / `USBD_BULK_Read()`. See also `USBD_BULK_GetNumBytesInBuffer()`.

A call of `USBD_BULK_Receive(Inst, NULL, 0, -1)` can be used to trigger an asynchronous read that stores the data into the internal buffer.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.14 USBD_BULK_ReadRecord()

Description

Reads data from host. The function reads up to NumBytes of data until a short packet has been received or after a timeout occurs. In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

Prototype

```
int USBD_BULK_ReadRecord(USB_BULK_HANDLE   hInst,
                         void            * pData,
                         unsigned          NumBytes,
                         int               Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Maximum number of bytes to read. Should be a multiple of the maximum packet size of the endpoint.
Timeout	Timeout given in milliseconds. A zero value results in an infinite timeout. If Timeout is -1, the read transfer will be initiated and the function returns immediately. In order to synchronize, USBD_BULK_WaitForRX() or USBD_BULK_PollForRX() needs to be called.

Return value

> 0	Number of bytes that have been read within the given timeout (if Timeout ≥ 0).
= 0	A timeout occurred (if Timeout > 0), a zero packet was received, the read is performed asynchronously (if Timeout < 0) or the target was disconnected during the function call and no data was read so far.
< 0	An error occurred.

Additional information

In case of a blocking (synchronous) use of the function (Timeout ≥ 0): If no error occurs, this function returns the number of bytes received. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return the number of bytes read so far. If the target was disconnected before this function was called, it returns USB_STATUS_ERROR.

In case of a non-blocking (asynchronous) use of the function (Timeout < 0): After synchronizing with USBD_BULK_WaitForRX() or USBD_BULK_PollForRX() the number of bytes read is 'NumBytes - USBD_BULK_GetNumBytesRemToRead(hInst)'.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.15 USBD_BULK_ReceivePoll()

Description

Reads data from the host. The function blocks until any data has been received or a timeout occurs (if `Timeout` ≥ 0). In contrast to `USBD_BULK_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received. In contrast to `USBD_BULK_Receive()` this function will continue the read transfer asynchronously in case of a timeout.

Prototype

```
int USBD_BULK_ReceivePoll(USB_BULK_HANDLE   hInst,
                          void            * pData,
                          unsigned          NumBytes,
                          unsigned          Timeout);
```

Parameters

Parameter	Description
`hInst`	Handle to a valid BULK instance, returned by `USBD_BULK_Add()`.
`pData`	Pointer to a buffer where the received data will be stored.
`NumBytes`	Maximum number of bytes to read.
`Timeout`	`Timeout` in milliseconds. 0 means infinite.

Return value

> 0	Number of bytes that have been read.
= 0	A timeout occurred (if `Timeout` > 0), zero packet received (not every controller supports this!) or the target was disconnected during the function call and no data was read so far.
< 0	Error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USBD_BULK_ReceivePoll()` will return as much data as is currently available -- up to the size of the buffer specified. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return `USB_STATUS_ERROR`.

If a read transfer was pending while the function is called, it returns `USB_STATUS_EP_BUSY`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBD_AddEP()`. This data can be retrieved by a later call to `USBD_BULK_Receive()` / `USBD_BULK_Read()`. See also `USBD_BULK_GetNumBytesInBuffer()`.

If a timeout occurs, the read transfer is not affected. Data sent from the host after the timeout is stored into the internal buffer of the endpoint and can be read by later calls to `USBD_BULK_ReceivePoll()`.

If `Timeout` = 0, the function behaves like `USBD_BULK_Receive()`.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.16 USBD_BULK_SetContinuousReadMode()

Description

Enables continuous read mode for the RX endpoint. In this mode every finished read transfer will automatically trigger another read transfer, as long as there is enough space in the internal buffer to receive another packet.

Prototype

```
void USBD_BULK_SetContinuousReadMode(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Additional information

To check how many bytes have been read into the buffer, the function USBD_BULK_GetNumBytesInBuffer() may be called. In order to read the data the function USBD_BULK_Receive() needs to be called (non-blocking).

The USB stack will use the buffer that was provided by the application with USBD_AddEP(). The transfer speed may be improved, if this buffer has a size of at least 2 * MaxPacketSize. Normally MaxPacketSize for full-speed devices is 64 bytes and for high-speed devices 512 bytes.

Example

```
USBD_BULK_SetContinuousReadMode(hInst);
<...>
for(;;) {
  //
  // Fetch data that was already read (non-blocking).
  //
  NumBytesReceived = USBD_BULK_Receive(hInst, &ac[0], sizeof(ac), -1);
  if (NumBytesReceived > 0) {
    //
    // We got some data
    //
    <.. Process data..>
  } else {
    <.. Nothing received yet, do application processing..>
  }
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.17 USBD_BULK_SetOnSetupRequest()

Description

Sets a callback function that is called when any setup request is sent from the host.

Prototype

```
void USBD_BULK_SetOnSetupRequest(USB_BULK_HANDLE   hInst,
                                 USB_ON_SETUP    * pfOnSetupRequest);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pfOnSetupRequest	Pointer to the callback function.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.18 USBD_BULK_SetOnRXEvent()

Description

Sets a callback function for the OUT endpoint that will be called on every RX event for that endpoint.

Prototype

```
void USBD_BULK_SetOnRXEvent(USB_BULK_HANDLE           hInst,
                            USB_EVENT_CALLBACK      * pEventCb,
                            USB_EVENT_CALLBACK_FUNC * pfEventCb,
                            void                    * pContext);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure. The structure is initialized by this function.
pfEventCb	Pointer to the callback routine that will be called on every event on the USB endpoint.
pContext	A pointer which is used as parameter for the callback function.

Additional information

The USB_EVENT_CALLBACK structure is private to the USB stack. It will be initialized by USBD_BULK_SetOnRXEvent(). The USB stack keeps track of all event callback functions using a linked list. The USB_EVENT_CALLBACK structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a read operation was started using one of the USBD_BULK_Read...() functions.

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
Events	A bit mask indicating which events occurred on the endpoint.
pContext	The pointer which was provided to the USBD_SetOnEvent() function.

Note that the callback function will be called within an ISR; therefore, it should never block. The first parameter to the callback function will contain a bit mask for all events that triggered the call:

Event	Description
USB_EVENT_DATA_READ	Some data was received from the host on the endpoint.
USB_EVENT_READ_COMPLETE	The last read operation was completed.
USB_EVENT_READ_ABORT	A read transfer was aborted.

 © 2010-2026 SEGGER Microcontroller GmbH

Example

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
  unsigned NumBytes;

  if (Events & USB_EVENT_DATA_READ) {
    NumBytes = USBD_BULK_GetNumBytesInBuffer(hInst);
    if (NumBytes) {
      r = USBD_BULK_Receive(hInst, Buff, NumBytes, -1);
      if (r > 0) {
        <.. process data in Buff..>
      }
    }
  }
}
// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBD_BULK_SetOnRXEvent(hInst, &_usb_callback, _OnEvent, NULL);
USBD_BULK_SetContinuousReadMode(hInst);
// Trigger first read
USBD_BULK_Receive(Inst, NULL, 0, -1);
<.. do anything else here while the data is processed in the callback ..>
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.19 USBD_BULK_SetOnTXEvent()

Description

Sets a callback function for the IN endpoint that will be called on every TX event for that endpoint.

Prototype

```
void USBD_BULK_SetOnTXEvent(USB_BULK_HANDLE           hInst,
                            USB_EVENT_CALLBACK      * pEventCb,
                            USB_EVENT_CALLBACK_FUNC * pfEventCb,
                            void                    * pContext);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure. The structure is initialized by this function.
pfEventCb	Pointer to the callback routine that will be called on every event on the USB endpoint.
pContext	A pointer which is used as parameter for the callback function.

Additional information

The USB_EVENT_CALLBACK structure is private to the USB stack. It will be initialized by USBD_BULK_SetOnTXEvent(). The USB stack keeps track of all event callback functions using a linked list. The USB_EVENT_CALLBACK structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a write operation was started using one of the USBD_BULK_Write…() functions.

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
Events	A bit mask indicating which events occurred on the endpoint.
pContext	The pointer which was provided to the USBD_SetOnEvent() function.

Note that the callback function will be called within an ISR, therefore it should never block. The first parameter to the callback function will contain a bit mask for all events that triggered the call:

Event	Description
USB_EVENT_DATA_SEND	Some data was sent to the host, so that (part of) the user write buffer may be reused by the application.
USB_EVENT_DATA_ACKED	Some data was acknowledged by the host.
USB_EVENT_WRITE_ABORT	A write transfer was aborted.
USB_EVENT_WRITE_COMPLETE	All write operations were completed.

 © 2010-2026 SEGGER Microcontroller GmbH

Example

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
  if ((Events & USB_EVENT_DATA_SEND) != 0 &&
        // Check for last write transfer to be completed.
        USBD_BULK_GetNumBytesRemToWrite(_hInst) == 0) {
            <.. prepare next data for writing..>
            // Send next packet of data.
            r = USBD_BULK_Write(_hInst, &ac[0], 200, -1);
            if (r < 0) {
              <.. error handling..>
            }
   }
}
// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBD_BULK_SetOnTXEvent(hInst, &_usb_callback, _OnEvent, NULL);
// Send the first packet of data using an asynchronous write operation.
r = USBD_BULK_Write(_hInst, &ac[0], 200, -1);
if (r < 0) {
  <.. error handling..>
}
<.. do anything else here while the whole data is sent..>
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.20 USBD_BULK_TxIsPending()

Description

Checks whether the TX (IN endpoint) is currently pending. Can be called in any context.

Prototype

```
int USBD_BULK_TxIsPending(USB_BULK_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().

Return value

1 We have queued data to be sent.
0 Queue is empty.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.21 USBD_BULK_WaitForRX()

Description

Waits (blocking) until the triggered USBD_BULK_ReadOverlapped() has received the desired data.

Prototype

```
int USBD_BULK_WaitForRX(USB_BULK_HANDLE hInst,
                        unsigned        Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
= 1 Timeout occurred.
< 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

Example

```
if (USBD_BULK_ReadOverlapped(hInst, &ac[0], 50) < 0) {
  <.. error handling..>
  return;
}
//
// USBD_BULK_ReadOverlapped() will return immediately.
// Do something else while data may be transferred.
//
<...>
//
// Now wait until we get all 50 bytes.
// USBD_BULK_WaitForRX() will block until a total of
// 50 bytes are read or a timeout occurs.
//
if (USBD_BULK_WaitForRX(hInst, timeout) != 0) {
  <.. timeout error handling..>
  return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.22 USBD_BULK_PollForRX()

Description

Waits (blocking) until the triggered `USBD_BULK_ReadOverlapped()` has received the desired data.

Prototype

```
int USBD_BULK_PollForRX(USB_BULK_HANDLE hInst,
                        unsigned        Timeout);
```

Parameters

Parameter	Description
`hInst`	Handle to a valid BULK instance, returned by `USBD_BULK_Add()`.
`Timeout`	`Timeout` in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
= 1 `Timeout` occurred.
< 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example

```
if (USBD_BULK_ReadOverlapped(hInst, &ac[0], 50) < 0) {
  <.. error handling..>
  return;
}
//
// USBD_BULK_ReadOverlapped() will return immediately.
// While waiting for the data, we will blink a LED with 200 ms interval.
// USBD_BULK_PollForRX() will return, if all data were read or 100 ms expired.
//
while ((r = USBD_BULK_PollForRX(hInst, 100)) > 0) {
  ToggleLED();
}
if (r < 0) {
  <.. error handling..>
  return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.23 USBD_BULK_WaitForTX()

Description

Waits (blocking) until a pending asynchronous USBD_BULK_Write() (Timeout parameter = -1) has sent the desired data.

Prototype

```
int USBD_BULK_WaitForTX(USB_BULK_HANDLE hInst,
                        unsigned        Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
= 1 Timeout occurred.
< 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 136).

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.24 USBD_BULK_PollForTX()

Description

Waits (blocking) until a pending asynchronous USBD_BULK_Write() (Timeout parameter = -1) has sent the desired data.

Prototype

```
int USBD_BULK_PollForTX(USB_BULK_HANDLE hInst,
                        unsigned        Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
= 1 Timeout occurred.
< 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example

```
if (USBD_BULK_Write(hInst, &ac[0], 50, -1) < 0) {
  <.. error handling..>
  return;
}
//
// USBD_BULK_Write() will return immediately.
// While waiting for the data to be transferred, we will blink a LED with
 200 ms interval.
// USBD_BULK_PollForTX() will return, if all data were sent or 100 ms expired.
//
while ((r = USBD_BULK_PollForTX(hInst, 100)) > 0) {
  ToggleLED();
}
if (r < 0) {
  <.. error handling..>
  return;
}
// Now all data have been sent.
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.25 USBD_BULK_WaitForTXReady()

Description

Waits (blocking) until the TX queue can accept another data packet. This function is used in combination with a non-blocking call to USBD_BULK_Write() , it waits until a new asynchronous write data transfer will be accepted by the USB stack.

Prototype

```
int USBD_BULK_WaitForTXReady(USB_BULK_HANDLE hInst,
                             int             Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is negative, the function will return immediately.

Return value

= 0 A new asynchronous write data transfer will be accepted.
= 1 The write queue is full, a call to USBD_BULK_Write() would return USB_STATUS_EP_BUSY.
< 0 Error occurred.

Additional information

If Timeout is 0, the function never returns 1.

If Timeout is -1, the function will not wait, but immediately return the current state.

Example

```
// Always keep the write queue full for maximum send speed.
for (;;) {
  pData = GetNextData(&NumBytes);
  // Wait until stack can accept a new write.
  USBD_BULK_WaitForTxReady(hInst, 0);
  // Issue write transfer.
  if (USBD_BULK_Write(hInst, pData, NumBytes, -1) < 0) {
    <.. error handling..>
  }
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.26 USBD_BULK_Write()

Description

Sends data to the USB host. Depending on the Timeout parameter, the function blocks until NumBytes have been written or a timeout occurs.

Prototype

```
int USBD_BULK_Write(      USB_BULK_HANDLE   hInst,
                    const void            * pData,
                          unsigned          NumBytes,
                          int               Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pData	Data that should be written.
NumBytes	Number of bytes to write.
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0	Successfully started an asynchronous write transfer or a timeout has occurred and no data was written.
> 0 && < NumBytes	Number of bytes that have been written before a timeout occurred.
= NumBytes	Write transfer successfully completed.
< 0	Error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

The USB stack is able to queue a small number of asynchronous write transfers (when using Timeout = -1). If a write transfer is still in progress when this function is called and the USB stack can not accept another write transfer request, the functions returns USB_STATUS_EP_BUSY. A synchronous write transfer (Timeout ≥ 0) will always block until the transfer (including all pending transfers) are finished or a timeout occurs.

In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 136).

In order to synchronize, USBD_BULK_WaitForTX() needs to be called. Another synchronization method would be to periodically call USBD_BULK_GetNumBytesRemToWrite() in order to see how many bytes still need to be written (this method is preferred when a non-blocking solution is necessary). The write operation can be canceled using USBD_BULK_CancelWrite().

If pData = NULL and NumBytes = 0, a zero-length packet is sent to the host.

The content of the buffer pointed to by pData must not be changed until the transfer has been completed.

Example

```
NumBytesWritten = USBD_BULK_Write(hInst, &ac[0], DataSize, 500);
if (NumBytesWritten <= 0) {
  <.. error handling..>
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
if (NumBytesWritten < DataSize) {
  <.. timeout occurred, data partially written within 500ms ..>
} else {
  <.. write completed successfully..>
}
```

See also *USBD_BULK_GetNumBytesRemToWrite* on page 157.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.27 USBD_BULK_WriteAsync()

Description

Sends data to the host asynchronously. The function does not block. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_BULK_WriteAsync(USB_BULK_HANDLE        hInst,
                          USB_ASYNC_IO_CONTEXT * pContext,
                          I8                     Send0PacketIfRequired);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pContext	Pointer to a structure of type USB_ASYNC_IO_CONTEXT containing parameters and a pointer to the callback function.
Send0PacketIfRequired	Specifies that a zero-length packet shall be sent when the last data packet is a multiple of MaxPacketSize.

Example

```
static void _AsyncCb(USB_ASYNC_IO_CONTEXT * pIOContext) {
  U8  *p;

  p = (U8 *)pIOContext->pContext;
  *p = 1;
}

<...>

USB_ASYNC_IO_CONTEXT IOContext;
U8                   AsyncComplete;

IOContext.NumBytesToTransfer = 5000;
IOContext.pData              = pBuff;
IOContext.pfOnComplete       = _AsyncCb;
IOContext.pContext           = (void *)&AsyncComplete;
AsyncComplete = 0;
USBD_BULK_WriteAsync(hInst, &IOContext, 1);
while (AsyncComplete == 0) {
  <.. Do other work. ..>
}
// Transaction is complete.
if (IOContext.Status < 0 || IOContext.NumBytesTransferred != 5000) {
  <.. error handling ..>
} else {
  <.. data written successfully ..>
}
<...>
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.2.28 USBD_BULK_WriteEx()

Description

Send data to the USB host with NULL packet control. This function behaves exactly like USBD_BULK_Write(). Additionally sending of a zero length packet after sending the data can be suppressed by setting Send0PacketIfRequired = 0.

Prototype

```
int USBD_BULK_WriteEx(      USB_BULK_HANDLE   hInst,
                      const void            * pData,
                            unsigned          NumBytes,
                            I8                Send0PacketIfRequired,
                            int               Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid BULK instance, returned by USBD_BULK_Add().
pData	Pointer to a buffer that contains the written data.
NumBytes	Number of bytes to write.
Send0PacketIfRequired	Specifies that a zero-length packet shall be sent when the last data packet is a multiple of MaxPacketSize. Normally MaxPacketSize for full-speed devices is 64 bytes. For high-speed devices the normal packet size is between 64 and 512 bytes.
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0	Successfully started an asynchronous write transfer or a timeout has occurred and no data was written.
> 0 && < NumBytes	Number of bytes that have been written before a timeout occurred.
= NumBytes	Write transfer successfully completed.
< 0	Error occurred.

Additional information

Normally USBD_BULK_Write() is called to let the stack send the data to the host and send an optional zero-length packet to tell the host that this was the last packet. This is the case when the last packet sent is MaxPacketSize bytes in size. When using this function, the zero-length packet handling can be controlled. This means the function can be called when sending data in multiple steps.

Example

```
// for high-speed devices
USBD_BULK_Write(hInst, _aBuffer1, 512, 0);
USBD_BULK_Write(hInst, _aBuffer2, 512, 0);
USBD_BULK_Write(hInst, _aBuffer3, 512, 0);
// this will send 6 packets to the host with sizes: 512, 0, 512, 0, 512, 0
USBD_BULK_WriteEx(hInst, _aBuffer1, 512, 0, 0);
USBD_BULK_WriteEx(hInst, _aBuffer2, 512, 0, 0);
USBD_BULK_WriteEx(hInst, _aBuffer3, 512, 1, 0);
// this will send 4 packets to the host with sizes: 512, 512, 512, 0
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.3 Data structures

5.4.3.1 USB_BULK_INIT_DATA

Description

Initialization structure that is needed when adding a BULK interface to emUSB-Device.

Type definition

```
typedef struct {
  U8   EPIn;
  U8   EPOut;
} USB_BULK_INIT_DATA;
```

Structure members

Member	Description
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.

 © 2010-2026 SEGGER Microcontroller GmbH

5.4.3.2 USB_BULK_INIT_DATA_EX

Description

Initialization structure that is needed when adding a BULK interface to emUSB-Device.

Type definition

```
typedef struct {
  U16           Flags;
  U8            EPIn;
  U8            EPOut;
  const char  * pInterfaceName;
  U8            InterfaceClass;
  U8            InterfaceSubClass;
  U8            InterfaceProtocol;
} USB_BULK_INIT_DATA_EX;
```

Structure members

Member	Description
Flags	Various flags. Valid bits: • USB_BULK_FLAG_USE_CUSTOM_CLASS_IDS - Allows to set custom values for the bInterfaceClass, bInterfaceSubClass and bInterfaceProtocol.
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
pInterfaceName	Name of the interface.
InterfaceClass	Only used when Flags has the USB_BULK_FLAG_USE_CUSTOM_CLASS_IDS bit set. Allows to set the USB Class ID to a different value (default is 0xFF).
InterfaceSubClass	Only used when Flags has the USB_BULK_FLAG_USE_CUSTOM_CLASS_IDS bit set. Allows to set the USB SubClass ID to a different value (default is 0xFF).
InterfaceProtocol	Only used when Flags has the USB_BULK_FLAG_USE_CUSTOM_CLASS_IDS bit set. Allows to set the USB Protocol ID to a different value (default is 0xFF).

 © 2010-2026 SEGGER Microcontroller GmbH

5.5 C Host API

To communicate with the target USB-Bulk stack, an API is provided that can be used on Windows, Linux, and macOS systems. This chapter describes the functions of the C API that can be used with the host system.

> **Note**
>
> There is also a C++ API that can be used to communicate with a target running the BULK class, see *C++ Host API*

To have an easy start-up when writing an application on the host side, these API functions have a simple interface and handle all required operations to communicate with the target USB-Bulk stack.

To use the API in an application the header file `USBBULK.h` must be included. Depending on the host operating system used the following components must be added to the host application:

- Windows: `USBBULK.lib` and `USBBULK.dll` (These files are provided for 32- and 64-bit applications).
- Linux: `USBBULK_Linux.c`.
- macOS: `USBBULK_MacOSX.c`.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.1 Bulk Host API list

The functions below are available on the host (PC) side.

Function	Description
USB-Bulk Basic functions	
USBBULK_Init()	This function needs to be called first.
USBBULK_Exit()	This is a cleanup function, it shall be called when exiting the application.
USBBULK_AddAllowedDeviceItem()	Adds the Vendor and Product ID to the list of devices the USBBULK API should look for.
USBBULK_GetNumAvailableDevices()	Returns the number of connected USB-Bulk devices.
USBBULK_Open()	Opens an existing device.
USBBULK_Close()	Closes an opened device.
USB-Bulk direct input/output functions	
USBBULK_Read()	Reads data from target device running emUSB-Device-Bulk.
USBBULK_ReadTimed()	Reads data from target device running emUSB-Device-Bulk within a given time-out.
USBBULK_Write()	Writes data to the device.
USBBULK_WriteTimed()	Writes data to the device within a given timeout.
USBBULK_CancelRead()	This cancels an initiated read.
USBBULK_FlushRx()	Flush the any received data.
USB-Bulk Control functions	
USBBULK_SetMode()	Sets the read and write mode for a specified device running emUSB-Device-Bulk.
USBBULK_GetMode()	Returns the current mode of the device.
USBBULK_SetReadTimeout()	Sets the default read timeout for an opened device.
USBBULK_SetWriteTimeout()	Sets a default write timeout for an opened device.
USBBULK_ResetINPipe()	Resets the IN pipe that is opened to the device.
USBBULK_ResetOUTPipe()	Resets the OUT pipe that is opened to the device.
USBBULK_ResetDevice()	Resets the device via a USB reset.
USB-Bulk general GET functions	
USBBULK_GetVersion()	Returns the version number of the USB-BULK API.
USBBULK_GetDevInfo()	Retrieves information about an opened USBBULK device.
USBBULK_GetDevInfoByIdx()	Retrieves information about a USB device.
USBBULK_GetUSBId()	Returns the Product and Vendor ID of an opened device.
USBBULK_GetProductName()	Retrieves the device/product name if available.

 © 2010-2026 SEGGER Microcontroller GmbH

Function	Description
USBBULK_GetVendorName()	Retrieves the vendor name of an opened USBBULK device.
USBBULK_GetSN()	Retrieves the USB serial number as a string which was sent by the device during the enumeration.
USBBULK_GetConfigDescriptor()	Gets the received target USB configuration descriptor of a specified device.
Data structures	
USBBULK_DEV_INFO	

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.2 USB-Bulk basic functions

5.5.2.1 USBBULK_Init()

Description

This function needs to be called first. This makes sure to have all structures and thread have been initialized. It also sets a callback in order to be notified when a device is added or removed.

Prototype

```
void USBBULK_Init(USBBULK_NOTIFICATION_FUNC * pfNotification,
                  void                      * pContext);
```

Parameters

Parameter	Description
pfNotification	Pointer to the user callback.
pContext	Context data that shall be called with the callback function.

Example

```
/*********************************************************************
*
*       _OnDevNotify
*
*  Function description:
*    Is called when a new device is found or an existing device is removed.
*
*  Parameters:
*    pContext  - Pointer to a context given when USBBULK_Init is called
*    Index     - Device Index that has been added or removed.
*    Event     - Type of event, currently the following are available:
*                  USBBULK_DEVICE_EVENT_ADD
*                  USBBULK_DEVICE_EVENT_REMOVE
*
*/
static void _OnDevNotify(void * pContext,
                         unsigned Index,
                         USBBULK_DEVICE_EVENT Event) {
  switch(Event) {
  case USBBULK_DEVICE_EVENT_ADD:
    printf("The following DevIndex has been added: %d", Index);
    NumDevices = USBBULK_GetNumAvailableDevices(&DeviceMask);
    break;
  case USBBULK_DEVICE_EVENT_REMOVE:
    printf("The following DevIndex has been removed: %d", Index);
    NumDevices = USBBULK_GetNumAvailableDevices(&DeviceMask);
    break;
  }
}
void MainTask(void) {
<...>
  USBBULK_Init(_OnDevNotify, NULL);
<...>
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.2.2 USBBULK_Exit()

Description

This is a cleanup function, it shall be called when exiting the application.

Prototype

```
void USBBULK_Exit(void);
```

Additional information

We recommend to call this function before exiting the application in order to remove all handles and resources that have been allocated.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.2.3 USBBULK_AddAllowedDeviceItem()

Description

Adds the Vendor and Product ID to the list of devices the USBBULK API should look for.

Prototype

```
void USBBULK_AddAllowedDeviceItem(uint16_t VendorId,
                                  uint16_t ProductId);
```

Parameters

Parameter	Description
VendorId	The desired Vendor ID mask that shall be used with the USBBULK API.
ProductId	The desired Product ID mask that shall be used with the USBBULK API.

Additional information

It is necessary to call this function first before calling `USBBULK_GetNumAvailableDevices()` or opening any connection to a device.

The function can be called multiple times to handle more than one pair of Vendor and Product IDs with the API.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.2.4 USBBULK_GetNumAvailableDevices()

Description

Returns the number of connected USB-Bulk devices.

Prototype

```
unsigned USBBULK_GetNumAvailableDevices(uint32_t * pMask);
```

Parameters

Parameter	Description
pMask	Pointer to a uint32_t variable to receive the connected device mask. This parameter can be NULL.

Return value

Number of available devices running emUSB-Device-Bulk.

Additional information

For each emUSB-Device device that is connected, a bit in pMask is set. For example if device 0 and device 2 are connected to the host, the value pMask points to will be 0x00000005.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.2.5 USBBULK_Open()

Description

Opens an existing device. The ID of the device can be retrieved by the function USBBULK_GetNumAvailableDevices() via the pDeviceMask parameter. Each bit set in the DeviceMask represents an available device. Currently 32 devices can be managed at once.

Prototype

```
USB_BULK_HANDLE USBBULK_Open(unsigned Id);
```

Parameters

Parameter	Description
Id	Device ID to be opened (0..31).

Return value

≠ 0 Handle to the opened device.
= 0 Error occurred.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.2.6 USBBULK_Close()

Description

Closes an opened device.

Prototype

```
void USBBULK_Close(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
hDevice	Handle to the device that shall be closed.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.3 USB-Bulk direct input/output functions

5.5.3.1 USBBULK_Read()

Description

Reads data from target device running emUSB-Device-Bulk.

Prototype

```
int USBBULK_Read(USB_BULK_HANDLE   hDevice,
                 void            * pBuffer,
                 int               NumBytes);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
pBuffer	Pointer to a buffer that shall receive the data.
NumBytes	Number of bytes to be read.

Return value

= NumBytes	All bytes have been successfully read.
> 0, < NumBytes	Number of bytes that have been read. If short read transfers are not allowed (normal mode) this indicates a timeout.
= 0	A timeout occurred, no data was read.
< 0	Error occurred.

Additional information

If short read transfers are allowed (see USBBULK_SetMode()) the function returns as soon as data is available, even if just a single byte was read. Otherwise the function blocks until NumBytes were read. In both cases the function returns if a timeout occurs. The default timeout used can be set with USBBULK_SetReadTimeout().

If NumBytes exceeds the maximum read size the driver can handle (the default value is 64 Kbytes), USBBULK_Read() will read the desired NumBytes in chunks of the maximum read size.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.3.2 USBBULK_ReadTimed()

Description

Reads data from target device running emUSB-Device-Bulk within a given timeout.

Prototype

```
int USBBULK_ReadTimed(USB_BULK_HANDLE   hDevice,
                      void            * pBuffer,
                      int               NumBytes,
                      unsigned          ms);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
pBuffer	Pointer to a buffer that shall receive the data.
NumBytes	Maximum number of bytes to be read.
ms	Timeout in milliseconds.

Return value

> 0	Number of bytes that have been read.
= 0	A timeout occurred during read.
< 0	Error, cannot read from the device.

Additional information

The function returns as soon as data is available, even if just a single byte was read. If no data is available, the functions return after the given timeout was expired.

If NumBytes exceeds the maximum read size the driver can handle (the default value is 64 Kbytes), USBBULK_ReadTimed() will read the desired NumBytes in chunks of the maximum read size.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.3.3 USBBULK_Write()

Description

Writes data to the device.

Prototype

```
int USBBULK_Write(      USB_BULK_HANDLE   hDevice,
                  const void            * pBuffer,
                        int               NumBytes);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
pBuffer	Pointer to a buffer that contains the data.
NumBytes	Number of bytes to be written. If NumBytes = 0, a zero length packet is written to the device.

Return value

= NumBytes	All bytes have been successfully written.
> 0, < NumBytes	Number of bytes that have been written.
= 0	A timeout occurred, no data was written.
< 0	Error, cannot write to the device.

Additional information

The function blocks until NumBytes were written or a timeout occurs. The default timeout used can be set with USBBULK_SetWriteTimeout().

If NumBytes exceeds the maximum write size the driver can handle (the default value is 64 Kbytes), USBBULK_Write() will write the desired NumBytes in chunks of the maximum write size.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.3.4 USBBULK_WriteTimed()

Description

Writes data to the device within a given timeout.

Prototype

```
int USBBULK_WriteTimed(      USB_BULK_HANDLE   hDevice,
                       const void            * pBuffer,
                             int               NumBytes,
                             unsigned          ms);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
pBuffer	Pointer to a buffer that contains the data.
NumBytes	Number of bytes to be written. If NumBytes = 0, a zero length packet is written to the device.
ms	Timeout in milliseconds.

Return value

= NumBytes	All bytes have been successfully written.
> 0, < NumBytes	Number of bytes that have been written.
= 0	A timeout occurred, no data was written.
< 0	Error, cannot write to the device.

Additional information

The function blocks until NumBytes were written or a timeout occurs.

If NumBytes exceeds the maximum write size the driver can handle (the default value is 64 Kbytes), USBBULK_WriteTimed() will write the desired NumBytes in chunks of the maximum write size.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.3.5 USBBULK_CancelRead()

Description

This cancels an initiated read.

Prototype

```
void USBBULK_CancelRead(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.

Additional information

Not supported on Linux and MacOSX.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.3.6 USBBULK_FlushRx()

Description

Flush the any received data.

Prototype

```
int USBBULK_FlushRx(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.

Return value

= 0 Error, bad handle.
≠ 0 Success, flushing the RX buffer was successful.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.4 USB-Bulk control functions

5.5.4.1 USBBULK_SetMode()

Description

Sets the read and write mode for a specified device running emUSB-Device-Bulk.

Prototype

```
unsigned USBBULK_SetMode(USB_BULK_HANDLE hDevice,
                         unsigned        Mode);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
Mode	Read and write mode for the USB-Bulk driver. This is a combination of the following flags, combined by binary or: • USBBULK_MODE_BIT_ALLOW_SHORT_READ • USBBULK_MODE_BIT_ALLOW_SHORT_WRITE

Return value

= 0 Operation failed (invalid handle).
≠ 0 The operation was successful.

Additional information

USBBULK_MODE_BIT_ALLOW_SHORT_READ allows short read transfers. Short transfers are transfers of less bytes than requested. If this bit is specified, the read function USBBULK_Read() returns as soon as data is available, even if it is just a single byte.

USBBULK_MODE_BIT_ALLOW_SHORT_WRITE allows short write transfers. USBBULK_Write() and USBBULK_WriteTimed() return after writing the minimal amount of data (either NumBytes or the maximal write transfer size).

Example

```
static void _TestMode(USB_BULK_HANDLE hDevice) {
  unsigned  Mode;
  char    * pText;
  Mode = USBBULK_GetMode(hDevice);
  if (Mode & USBBULK_MODE_BIT_ALLOW_SHORT_READ) {
    pText = "USE_SHORT_MODE";
  } else {
    pText = "USE_NORMAL_MODE";
  }
  printf("USB-Bulk driver is in %s for device %d\n", pText, (int)hDevice);
  printf("Set mode to USBBULK_MODE_BIT_ALLOW_SHORT_READ\n");
  USBBULK_SetMode(hDevice, USBBULK_MODE_BIT_ALLOW_SHORT_READ);
  Mode = USBBULK_GetMode(hDevice);
  if (Mode & USBBULK_MODE_BIT_ALLOW_SHORT_READ) {
    pText = "USE_SHORT_MODE";
  } else {
    pText = "USE_NORMAL_MODE";
  }
  printf("USB-Bulk driver is now in %s for device %d\n", pText,(int)hDevice);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.4.2 USBBULK_GetMode()

Description

Returns the current mode of the device.

Prototype

```
unsigned USBBULK_GetMode(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.

Return value

A combination of the following flags, combined by binary or:

- USBBULK_MODE_BIT_ALLOW_SHORT_READ - Short read mode is enabled.
- USBBULK_MODE_BIT_ALLOW_SHORT_WRITE - Short write mode is enabled.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.4.3 USBBULK_SetReadTimeout()

Description

Sets the default read timeout for an opened device.

Prototype

```
void USBBULK_SetReadTimeout(USB_BULK_HANDLE hDevice,
                            int             Timeout);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
Timeout	Timeout in milliseconds.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.4.4 USBBULK_SetWriteTimeout()

Description

Sets a default write timeout for an opened device.

Prototype

```
void USBBULK_SetWriteTimeout(USB_BULK_HANDLE hDevice,
                             int             Timeout);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
Timeout	Timeout in milliseconds.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.4.5 USBBULK_ResetINPipe()

Description

Resets the IN pipe that is opened to the device. It also flushes any data the USB bulk driver would cache.

Prototype

```
int USBBULK_ResetINPipe(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.

Return value

≠ 0	The operation was successful.
= 0	Operation failed. Either an invalid handle was used or the pipe cannot be flushed.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.4.6 USBBULK_ResetOUTPipe()

Description

Resets the OUT pipe that is opened to the device.

Prototype

```
int USBBULK_ResetOUTPipe(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.

Return value

≠ 0	The operation was successful.
= 0	Operation failed. Either an invalid handle was used or the pipe cannot be flushed.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.4.7 USBBULK_ResetDevice()

Description

Resets the device via a USB reset. This can be used when the device does not work properly and may be reactivated via USB reset. This will force a re-enumeration of the device.

Prototype

```
int USBBULK_ResetDevice(USB_BULK_HANDLE hDevice);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.

Return value

≠ 0	The operation was successful.
= 0	Operation failed. Either an invalid handle was used or the device cannot be reset.

Additional information

After the device has been reset it is necessary to re-open the device as the current handle will become invalid.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.5 USB-Bulk general GET functions

5.5.5.1 USBBULK_GetVersion()

Description

Returns the version number of the USBBULK API.

Prototype

```
unsigned USBBULK_GetVersion(void);
```

Return value

Version number, format:

< Major Version><Minor Version><Subversion> (Mmmrr, decimal).

Example: 30203 is 3.02c

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.5.2 USBBULK_GetDevInfo()

Description

Retrieves information about an opened USBBULK device.

Prototype

```
void USBBULK_GetDevInfo(USB_BULK_HANDLE     hDevice,
                        USBBULK_DEV_INFO * pDevInfo);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
pDevInfo	Pointer to a device info structure of type USBBULK_DEV_INFO.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.5.3 USBBULK_GetDevInfoByIdx()

Description

Retrieves information about a USB device.

Prototype

```
int USBBULK_GetDevInfoByIdx(unsigned           Idx,
                            USBBULK_DEV_INFO * pDevInfo);
```

Parameters

Parameter	Description
Idx	Index of the device.
pDevInfo	Pointer to a device info structure of type USBBULK_DEV_INFO.

Return value

= 0; Error, bad device index.
≠ 0 Success

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.5.4 USBBULK_GetUSBId()

Description

Returns the Product and Vendor ID of an opened device.

Prototype

```
void USBBULK_GetUSBId(USB_BULK_HANDLE   hDevice,
                      uint16_t        * pVendorId,
                      uint16_t        * pProductId);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
pVendorId	Pointer to a variable that receives the Vendor ID.
pProductId	Pointer to a variable that receives the Product ID.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.5.5 USBBULK_GetProductName()

Description

Retrieves the device/product name if available.

Prototype

```
int USBBULK_GetProductName(USB_BULK_HANDLE   hDevice,
                           char            * sProductName,
                           unsigned          BufferSize);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
sProductName	Pointer to a buffer that should receive the string.
BufferSize	Size of the buffer, given in bytes.

Return value

= 0 Error, product name not available or buffer to small.
≠ 0 Success, product name stored in buffer pointed by sProductName as 0-terminated string.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.5.6 USBBULK_GetVendorName()

Description

Retrieves the vendor name of an opened USBBULK device.

Prototype

```
int USBBULK_GetVendorName(USB_BULK_HANDLE   hDevice,
                          char            * sVendorName,
                          unsigned          BufferSize);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
sVendorName	Pointer to a buffer that should receive the string.
BufferSize	Size of the buffer, given in bytes.

Return value

= 0 Error, bad handle.
≠ 0 Success, vendor name stored in buffer pointed by sVendorName as 0-terminated string.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.5.7 USBBULK_GetSN()

Description

Retrieves the USB serial number as a string which was sent by the device during the enumeration.

Prototype

```
int USBBULK_GetSN(USB_BULK_HANDLE   hDevice,
                  uint8_t         * pBuffer,
                  unsigned          BuffSize);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
pBuffer	Pointer to a buffer which shall receive the serial number of the device.
BuffSize	Size of the buffer in bytes.

Return value

= 0 Operation failed. Either an invalid handle was used or the serial number is not available.
≠ 0 The operation was successful.

Additional information

If the function succeeds, the buffer pointed by pBuffer contains the serial number of the device as 0-terminated string. If BuffSize is too small, the serial number is truncated.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.5.8 USBBULK_GetConfigDescriptor()

Description

Gets the received target USB configuration descriptor of a specified device.

Prototype

```
int USBBULK_GetConfigDescriptor(USB_BULK_HANDLE   hDevice,
                                void            * pBuffer,
                                int               Size);
```

Parameters

Parameter	Description
hDevice	Handle to the opened device.
pBuffer	Pointer to the buffer that shall store the descriptor.
Size	Size of the buffer, given in bytes.

Return value

≠ 0	Size of the returned USB configuration descriptor (Success).
= 0	Operation failed. Either an invalid handle was used or the buffer that shall store the config descriptor is too small.

 © 2010-2026 SEGGER Microcontroller GmbH

5.5.6 USB-Bulk data structures

5.5.6.1 USBBULK_DEV_INFO

Type definition

```
typedef struct _USBBULK_DEV_INFO {
  uint16_t   VendorId;
  uint16_t   ProductId;
  char       acSN[256];
  char       acDevName[255];
  uint8_t    InterfaceNo;
  uint8_t    Speed;
} USBBULK_DEV_INFO;
```

Structure members

Member	Description
VendorId	Vendor ID of the device.
ProductId	Product ID of the device.
acSN	0-terminated string which holds the serial number of the device.
acDevName	0-terminated string which holds the device name.
InterfaceNo	Interface number used by this device.
Speed	Device speed. One of the following: USBBULK_SPEED_UNKNOWN USBBULK_SPEED_LOW USBBULK_SPEED_FULL USBBULK_SPEED_HIGH USBBULK_SPEED_SUPER

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 6

Vendor Specific Class (VSC)

This chapter describes how to get emUSB-Device-VSC up and running.



 © 2010-2026 SEGGER Microcontroller GmbH

6.1 Vendor Specific Class

The Vendor Specific Class (VSC) is located in the directory *USB*. All C files in the directory should be included in the project (compiled and linked as part of your project). The files in this directory are maintained by SEGGER and should not require any modification. All files requiring modifications have been placed in other directories.

 © 2010-2026 SEGGER Microcontroller GmbH

6.2 Requirements for the Host (PC)

In order to communicate with a target (client) running emUSB-Device, the operating system running on the host must recognize the device connected to it.

6.2.1 Windows

Microsoft's Windows operating system contains a generic driver called WinUSB.sys that is used to handle all communication to an emUSB-Device running a VSC interface. If an emUSB device is connected to a Windows 8, 8.1 and 10 PC for the first time, Windows will install the WinUSB driver automatically. For Windows versions less than Windows 8, Microsoft provides a driver for Windows Vista and Windows 7, but this needs to be installed manually. A driver installation tool including the mentioned driver is available in the `Windows\USB\VSC\WinUSBInstall`. In order to get emUSB VSC running with the WinUSB driver, the following must be considered:

- The function `USBD_VSC_SetMSDescInfo()` must be called in the target application.
- The Product IDs 1234 and 1121 must not be used.

6.2.2 Linux

Linux can handle emUSB VSC devices out of the box.

By default, a USB device can only be accessed by a process that is running with "root" rights. In order to use the USB VSC device from normal user programs, a `udev` rule has to be configured for the device (refer to the Linux `udev` documentation). The emUSB-Device release contains a sample configuration file `99-emUSBD.rules`, which may be modified and copied to /etc/udev/rules.d on the host machine.

6.2.3 macOS

macOS can handle emUSB VSC devices out of the box.

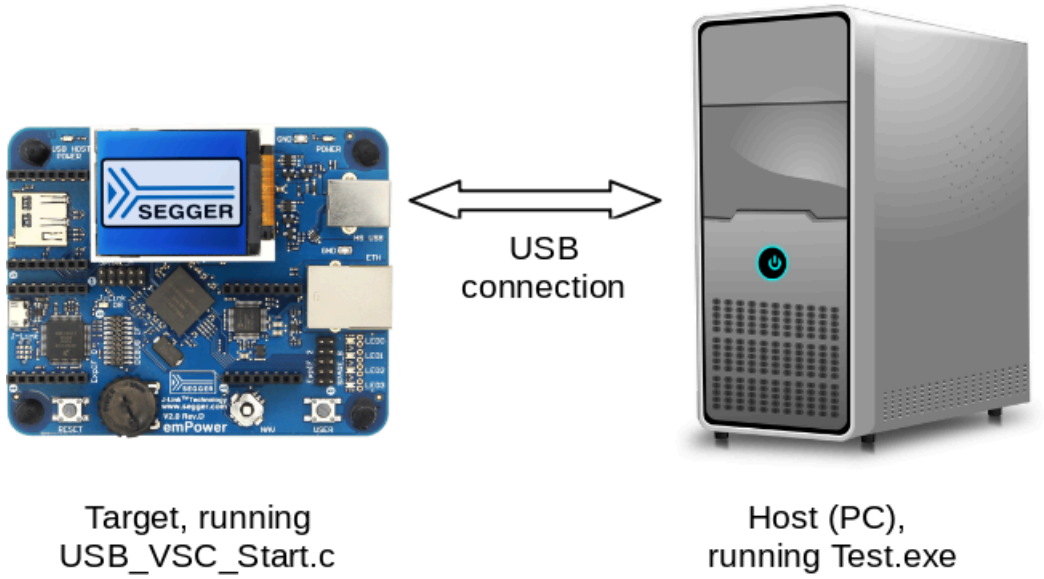
 © 2010-2026 SEGGER Microcontroller GmbH

6.3 Example application

Example applications for both the target (client) and the PC (host) are supplied. These can be used for testing the correct installation and proper function of the device running emUSB-Device.

The host sample applications can be used for Windows, Linux and macOS. Precompiled executables for Windows can be found in the subfolder `Windows/USB/Bulk/SampleApplication/Exe`.

The application `USB_VSC_Start.c` is a modified echo server; the application receives data, modifies the first byte and sends it back to the host. It also contains the functionality to measure USB transfer speed.



The example applications for the target-side are supplied in source code in the `Application` directory.

For information on how to compile the host examples (especially for Linux and macOS), refer to *Compiling the PC example application* on page 145.

The start application will of course later on be replaced by the real application program. For the purpose of getting emUSB-Device up and running as well as doing an initial test, the start application should not be modified.

6.3.1 Running the example applications

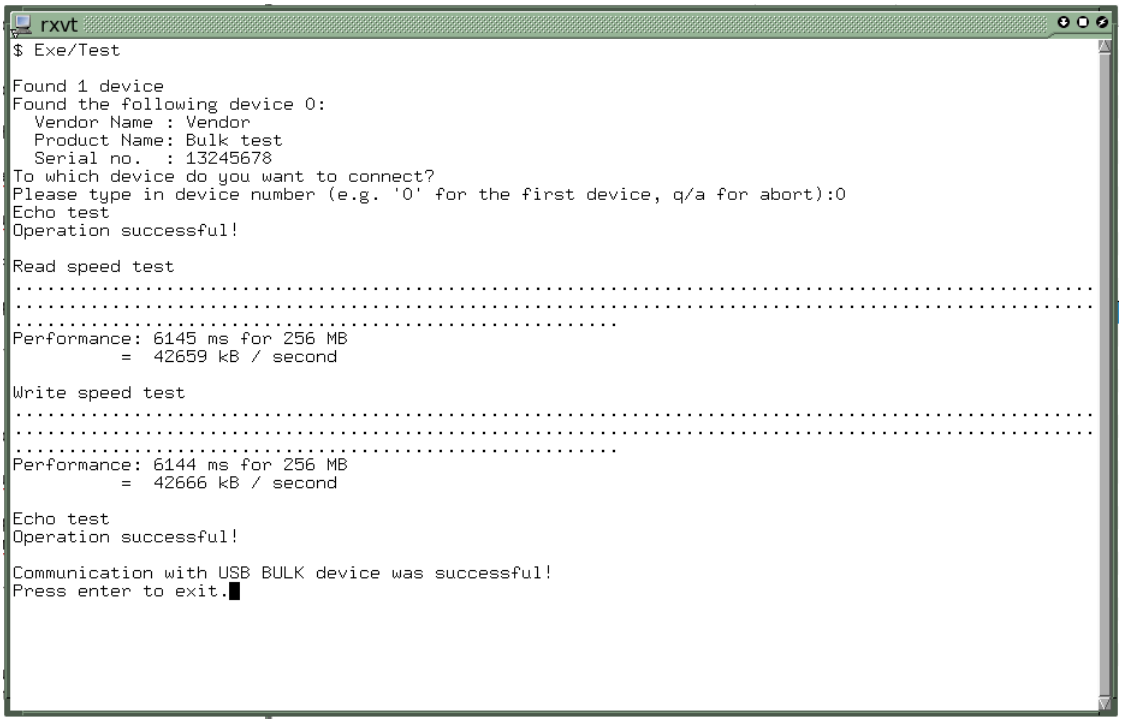
To test the emUSB-Device-VSC component, build and download the `USB_VSC_Start.c` for the target-side.

To run one of the example applications, simply start the executable `Test.exe`, for example by double-clicking it.

If a connection can be established, it exchanges data with the target, testing the USB connection.

Example output of `Test.exe`:

 © 2010-2026 SEGGER Microcontroller GmbH



6.3.2 Compiling the PC example application

6.3.2.1 Windows

For compiling the example application you need Visual C++ 2015 (or later).

The source code of the sample application is located in the subfolder `Windows/USB/BULK/USBVAPI_SampleApplication/Src`. Open the file `USBVAPI_Samples.sln` and compile the source.

6.3.2.2 Linux

The subfolder `Windows/USB/Bulk/USBVAPI_SampleApplication` contains a Makefile for Linux. Change to this folder and execute “make”.

6.3.2.3 macOS

The subfolder `Windows/USB/Bulk/USBVAPI_SampleApplication` contains a Makefile for macOS. Change to this folder and execute “make -f `Makefile_MacOS`”.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4 Target API

This chapter describes the functions that can be used with the target system.

General information

To communicate with the host, the sample application project includes USB-specific header and source files (`USB.h`, `USB_Main.c`, `USB_Setup.c`, `USB_VSC.c`, `USB_VSC.h`). These files contain API functions to communicate with the USB host through the emUSB-Device driver.

Purpose of the USB Device API functions

To have an easy startup when writing an application on the device side, these API functions have a simple interface and handle all operations that need to be done to communicate with the host's kernel.

Therefore, all operations that need to write to or read from the emUSB-Device are handled internally by the provided API functions.

6.4.1 Target interface function list

Routine	Explanation
USB VSC functions	
`USBD_VSC_Add()`	Adds a VSC interface to emUSB-Device.
`USBD_VSC_AddAlternateInterface()`	Adds an alternative interface for USB-VSC interface.
`USBD_VSC_CancelIO()`	Cancels any non-blocking/blocking data transfer operation that is pending.
`USBD_VSC_GetNumBytesInBuffer()`	Returns the number of bytes that are available in the internal BULK-OUT end-point buffer.
`USBD_VSC_GetNumBytesRemToRead()`	Get the number of remaining bytes to read by an active read operation.
`USBD_VSC_GetNumBytesRemToWrite()`	After starting a non-blocking write oper-ation this function can be used to period-ically check how many bytes still have to be written.
`USBD_VSC_Read()`	Reads data from the host with a given timeout.
`USBD_VSC_ReadAsync()`	Reads data from the host asynchronously.
`USBD_VSC_SetContinuousReadMode()`	Enables continuous read mode for the RX endpoint.
`USBD_VSC_SetOnSetupRequest()`	Sets a callback function that is called when any setup request is sent from the host.
`USBD_VSC_SetOnEPEvent()`	Sets a callback function for the IN end-point that will be called on every TX event for that endpoint.
`USBD_VSC_TxIsPending()`	Checks whether the TX (IN endpoint) is currently pending.
`USBD_VSC_WaitEP()`	Waits (blocking) until the triggered `USB-D_VSC_ReadOver lapped()` has received the desired data.
`USBD_VSC_PollEP()`	Waits (blocking) until the triggered `USB-D_VSC_ReadOver lapped()` has received the desired data.

 © 2010-2026 SEGGER Microcontroller GmbH

Routine	Explanation
USBD_VSC_StallEP()	Stall an EP.
USBD_VSC_WaitForTXReady()	Waits (blocking) until the TX queue can accept another data packet.
USBD_VSC_Write()	Sends data to the USB host.
USBD_VSC_WriteAsync()	Sends data to the host asynchronously.
USBD_VSC_SetOnVendorRequest()	Sets a callback function that is called when a setup vendor request is sent from the host to the specified interface index.
USBD_VSC_SetOnSetupRequest()	Sets a callback function that is called when any setup request is sent from the host.
USBD_VSC_SetOnClassRequest()	Sets a callback function that is called when a setup class request is sent from the host to the specified interface index.
USBD_VSC_AddAlternateInterface()	Adds an alternative interface for USB-VSC interface.
Data structures and callbacks	
USB_VSC_INIT_DATA	Initialization structure that is needed when adding a VSC interface to emUSB-Device.
USB_VSC_MSOSDESC_INFO	MS OS descriptor structure that contains for MS related OSes information how to deal with device with out having a driver store.
USB_VSC_ON_ADD_FUNCTION_DESC	Call back that is used to add an additional descriptor between the interface or one of its alternate setting descriptor and the endpoint descriptor(s).
USB_VSC_ON_SET_INTERFACE	Global callback function that is called whenever an alternate setting is set for an interface that was added with USBD_VSC_Add().

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2 USB-VSC functions

6.4.2.1 USBD_VSC_Add()

Description

Adds a VSC interface to emUSB-Device.

Prototype

```
USB_VSC_HANDLE USBD_VSC_Add(const USB_VSC_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to USB_VSC_INIT_DATA structure.

Return value

Handle to a valid VSC instance.

Example

Example excerpt from USB_VSC_Echo1.c:

Example

```
static void _AddVSC(void) {
  static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
  USB_VSC_INIT_DATA  Init;
  Init.Flags = 0;
  Init.EPIn  = USBD_AddEP(USB_DIR_IN,
                          USB_TRANSFER_TYPE_BULK,
                          USB_HS_BULK_MAX_PACKET_SIZE,
                          NULL,
                          0);
  Init.EPOut = USBD_AddEP(USB_DIR_OUT,
                          USB_TRANSFER_TYPE_BULK,
                          USB_HS_BULK_MAX_PACKET_SIZE,
                          _abOutBuffer,
                          USB_HS_BULK_MAX_PACKET_SIZE);
  Init.pInterfaceName = "VSC Interface";
  USBD_VSC_Add(&Init);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.2 USBD_VSC_AddAlternateInterface()

Description

Adds an alternative interface for USB-VSC interface.

Prototype

```
void USBD_VSC_AddAlternateInterface(      USB_VSC_HANDLE      hInst,
                                    const USB_VSC_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
hInst	Handle to a valid VSC instance, returned by USBD_VSC_Add-d().
pInitData	Pointer to USB_VSC_INIT_DATA structure.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.3 USBD_VSC_CancelIO()

Description

Cancels any non-blocking/blocking data transfer operation that is pending.

Prototype

```
void USBD_VSC_CancelIO(U8 EPIndex);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().

Additional information

This function shall be called when a pending asynchronous transfer operation should be canceled. The function can be called from any task. In case of canceling a blocking operation, this function must be called from another task.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.4 USBD_VSC_GetNumBytesInBuffer()

Description

Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer. This function does not start a read transfer.

Prototype

```
unsigned USBD_VSC_GetNumBytesInBuffer(U8 EPIndex);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().

Return value

Number of bytes that are available in the internal BULK-OUT endpoint buffer.

Additional information

If the host is sending more data than your target application has requested, the remaining data will be stored in an internal buffer. This function shows how many bytes are available in this buffer.

The number of bytes returned by this function can be read using USBD_VSC_Read() without blocking.

Example

Your host application sends 50 bytes. Your target application only requests to receive 1 byte. In this case, the target application will get 1 byte, and the remaining 49 bytes are stored in an internal buffer. When your target application now calls USBD_VSC_GetNumBytesIn-Buffer(), it will return the number of bytes that are available in the internal buffer (49).

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.5 USBD_VSC_GetNumBytesRemToRead()

Description

Get the number of remaining bytes to read by an active read operation. This function is to be used in combination with `USBD_VSC_Read()` where Timeout=-1. After starting the read operation this function can be used to periodically check how many bytes still have to be read.

Prototype

```
unsigned USBD_VSC_GetNumBytesRemToRead(U8 EPIndex);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to `USBD_VSC_Add()`.

Return value

≥ 0 Number of bytes which have not yet been read.
< 0 Error occurred.

Additional information

Alternatively the blocking function `USBD_VSC_WaitForRX()` can be used.

Example

```
NumBytesReceived = USBD_VSC_Read(hInst, &ac[0], 50, -1, 0);
if (NumBytesReceived < 0) {
  <.. error handling..>
}
if (NumBytesReceived > 0) {
  // Already had some data in the internal buffer.
  // The first 'NumBytesReceived' bytes may be processed here.
  <...>
} else {
  // Wait until we get all 50 bytes
  while (USBD_VSC_GetNumBytesRemToRead(hInst) > 0) {
    USB_OS_Delay(50);
  }
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.6 USBD_VSC_GetNumBytesRemToWrite()

Description

After starting a non-blocking write operation this function can be used to periodically check how many bytes still have to be written.

Prototype

```
unsigned USBD_VSC_GetNumBytesRemToWrite(U8 EPIndex);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().

Return value

Number of bytes which have not yet been written.

Additional information

Alternatively the blocking function USBD_VSC_WaitForTX() can be used.

Example

```
r = USBD_VSC_Write(hInst, &ac[0], TRANSFER_SIZE, -1);
if (r < 0) {
  <.. error handling..>
}
// NumBytesToWrite shows how many bytes still have to be written.
while (USBD_VSC_GetNumBytesRemToWrite(hInst) > 0) {
  USB_OS_Delay(50);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.7 USBD_VSC_Read()

Description

Reads data from the host with a given timeout.

Prototype

```
int USBD_VSC_Read(U8          EPIndex,
                  void      * pData,
                  unsigned    NumBytes,
                  int         Timeout,
                  unsigned    Flags);
```

Parameters

Parameter	Description
EPIndex	One of the EPIndex was used in pInitData when calling USBD_VSC_Add().
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.
Flags	Various flags: • USB_VSC_READ_FLAG_RECEIVE - This turns the read function into the receive mode. • USB_VSC_READ_FLAG_POLL - Can only be used with the USB_VSC_READ_FLAG_RECEIVE. This function will not abort the transfer in case of a timeout.

Return value

= NumBytes	Requested data was successfully read within the given timeout.
≥ 0 && < NumBytes	Timeout has occurred. Number of bytes that have been read within the given timeout.
< 0	Error occurred.

Additional information

- Normal mode:

This function blocks a task until all data have been read or a timeout expires. This function also returns when the device is disconnected from host or when a USB reset occurs.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via USBD_AddEP(). This data can be retrieved by a later call to USBD_VSC_Read(). See also USBD_VSC_GetNumBytesInBuffer().

When Timeout = -1 the read transfer will be initiated and the function returns immediately. In order to synchronize, USBD_VSC_WaitForRX() needs to be called. Alternatively the function USBD_VSC_GetNumBytesRemToRead() can be called periodically to check whether all bytes have been read or not. The read operation can be canceled using USBD_VSC_Cancel(). The buffer pointed to by pData must be valid until the read operation is terminated.

In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

If a read transfer was still pending while the function is called, it returns USB_STATUS_EP_BUSY.

- Receive mode:

If no error occurs, this function returns the number of bytes received. Calling USBD_VSC_Read() will return as much data as is currently available -- up to the size of the

 © 2010-2026 SEGGER Microcontroller GmbH

buffer specified. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return USB_STATUS_ERROR.

If a read transfer was pending while the function is called, it returns USB_STATUS_EP_BUSY.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via USBD_AddEP(). This data can be retrieved by a later call to USBD_VSC_Read(). See also USBD_VSC_GetNumBytesInBuffer().

A call of USBD_VSC_Read(EPIndex, NULL, 0, -1, USB_VSC_) can be used to trigger an asynchronous read that stores the data into the internal buffer.

- Receive in polled mode:

In contrast to receive mode this function will continue the read transfer asynchronously in case of a timeout.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.8 USBD_VSC_ReadAsync()

Description

Reads data from the host asynchronously. The function does not wait for the data to be received. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_VSC_ReadAsync(U8                      EPIndex,
                        USB_ASYNC_IO_CONTEXT * pContext,
                        int                    ShortRead);
```

Parameters

Parameter	Description
EPIndex	A valid OUT EP Index that was also passed to USBD_VSC_Add().
pContext	Pointer to a structure of type USB_ASYNC_IO_CONTEXT containing parameters and a pointer to the callback function.
ShortRead	• 0: The transfer is completed successfully after all bytes have been read. • 1: The transfer is completed successfully after one packet has been read.

Example

```
static void _AsyncCb(USB_ASYNC_IO_CONTEXT * pIOContext) {
  U8  *p;

  p = (U8 *)pIOContext->pContext;
  *p = 1;
}

<...>

USB_ASYNC_IO_CONTEXT IOContext;
U8                   AsyncComplete;

IOContext.NumBytesToTransfer = 5000;
IOContext.pData              = pBuff;
IOContext.pfOnComplete       = _AsyncCb;
IOContext.pContext           = (void *)&AsyncComplete;
AsyncComplete = 0;
USBD_VSC_ReadAsync(hInst, &IOContext, 0);
while (AsyncComplete == 0) {
  <.. Do other work. ..>
}
// Transaction is complete.
if (IOContext.Status < 0 || IOContext.NumBytesTransferred != 5000) {
  <.. error handling ..>
} else {
  <.. Process the data ..>
}
<...>
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.9 USBD_VSC_SetContinuousReadMode()

Description

Enables continuous read mode for the RX endpoint. In this mode every finished read transfer will automatically trigger another read transfer, as long as there is enough space in the internal buffer to receive another packet.

Prototype

```
void USBD_VSC_SetContinuousReadMode(U8 EPIndex);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().

Additional information

To check how many bytes have been read into the buffer, the function USBD_VSC_GetNumBytesInBuffer() may be called. In order to read the data the function USBD_VSC_Read() needs to be called (non-blocking).

The USB stack will use the buffer that was provided by the application with USBD_AddEP(). The transfer speed may be improved, if this buffer has a size of at least 2 * MaxPacketSize. Normally MaxPacketSize for full-speed devices is 64 bytes and for high-speed devices 512 bytes.

Example

```
USBD_VSC_SetContinuousReadMode(hInst);
<...>
for(;;) {
  //
  // Fetch data that was already read (non-blocking).
  //
  NumBytesReceived = USBD_VSC_Read(hInst, &ac[0], sizeof(ac), -1, USB_VSC_READ_FLAG_POLL);
  if (NumBytesReceived > 0) {
    //
    // We got some data
    //
    <.. Process data..>
  } else {
    <.. Nothing received yet, do application processing..>
  }
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.10 USBD_VSC_SetOnSetupRequest()

Description

Sets a callback function that is called when any setup request is sent from the host.

Prototype

```
void USBD_VSC_SetOnSetupRequest(USB_VSC_HANDLE   hInst,
                                USB_ON_SETUP   * pfOnSetupRequest);
```

Parameters

Parameter	Description
hInst	Handle to a valid VSC instance, returned by USBD_VSC_Add-d().
pfOnSetupRequest	Pointer to the callback function.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.11 USBD_VSC_SetOnEPEvent()

Description

Sets a callback function for the IN endpoint that will be called on every TX event for that endpoint.

Prototype

```
void USBD_VSC_SetOnEPEvent(U8                          EPIndex,
                           USB_EVENT_CALLBACK      * pEventCb,
                           USB_EVENT_CALLBACK_FUNC * pfEventCb,
                           void                    * pContext);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure. The structure is initialized by this function.
pfEventCb	Pointer to the callback routine that will be called on every event on the USB endpoint.
pContext	A pointer which is used as parameter for the callback function.

Additional information

The USB_EVENT_CALLBACK structure is private to the USB stack. It will be initialized by USBD_VSC_SetOnEPEvent(). The USB stack keeps track of all event callback functions using a linked list. The USB_EVENT_CALLBACK structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a transfer operation was started using either USBD_VSC_Read() or USBD_VSC_Write() functions.

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
Events	A bit mask indicating which events occurred on the endpoint.
pContext	The pointer which was provided to the USBD_SetOnEvent() function.

Note that the callback function will be called within an ISR; therefore, it should never block. The first parameter to the callback function will contain a bit mask for all events that triggered the call:

EP Direction	Event	Description
OUT	USB_EVENT_DATA_READ	Some data was received from the host on the endpoint.
OUT	USB_EVENT_READ_COMPLETE	The last read operation was completed.
OUT	USB_EVENT_READ_ABORT	A read transfer was aborted.
IN	USB_EVENT_DATA_SEND	Some data was sent to the host, so that (part of) the user write buffer may be reused by the application.

 © 2010-2026 SEGGER Microcontroller GmbH

EP Direction	Event	Description
IN	USB_EVENT_DATA_ACKED	Some data was acknowledged by the host.
IN	USB_EVENT_WRITE_ABORT	A write transfer was aborted.
IN	USB_EVENT_WRITE_COMPLETE	All write operations were completed.

 © 2010-2026 SEGGER Microcontroller GmbH

Example for an OUT EP

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
  unsigned NumBytes;

  if (Events & USB_EVENT_DATA_READ) {
    NumBytes = USBD_VSC_GetNumBytesInBuffer(hInst);
    if (NumBytes) {
      r = USBD_VSC_Read(hInst, Buff, NumBytes, -1, USB_VSC_READ_FLAG_RECEIVE);
      if (r > 0) {
        <.. process data in Buff..>
      }
    }
  }
}
// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBD_VSC_SetOnEPEvent(EPOut, &_usb_callback, _OnEvent, NULL);
USBD_VSC_SetContinuousReadMode(EPOut);
// Trigger first read
USBD_VSC_Read(EPOut, NULL, 0, -1, USB_VSC_READ_FLAG_POLL);
<.. do anything else here while the data is processed in the callback ..>
```

 © 2010-2026 SEGGER Microcontroller GmbH

Example for an IN EP

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
  if ((Events & USB_EVENT_DATA_SEND) != 0 &&
        // Check for last write transfer to be completed.
USBD_VSC_GetNumBytesRemToWrite(_hInst) == 0) {
            <.. prepare next data for writing..>
            // Send next packet of data.
            r = USBD_VSC_Write(_hInst, &ac[0], 200, -1);
            if (r < 0) {
              <.. error handling..>
            }
   }
}
// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBD_VSC_SetOnEPEvent(hInst, &_usb_callback, _OnEvent, NULL);
// Send the first packet of data using an asynchronous write operation.
r = USBD_VSC_Write(_hInst, &ac[0], 200, -1, 0);
if (r < 0) {
  <.. error handling..>
}
<.. do anything else here while the whole data is sent..>
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.12 USBD_VSC_TxIsPending()

Description

Checks whether the TX (IN endpoint) is currently pending. Can be called in any context.

Prototype

```
int USBD_VSC_TxIsPending(U8 EPIndex);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().

Return value

1	We have queued data to be sent.
0	Queue is empty.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.13 USBD_VSC_WaitEP()

Description

Waits (blocking) until the triggered USBD_VSC_ReadOverlapped() has received the desired data.

Prototype

```
int USBD_VSC_WaitEP(U8       EPIndex,
                    unsigned Timeout);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
= 1 Timeout occurred.
< 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

</think>

Example for an IN EP

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
  if ((Events & USB_EVENT_DATA_SEND) != 0 &&
        // Check for last write transfer to be completed.
USBD_VSC_GetNumBytesRemToWrite(_hInst) == 0) {
            <.. prepare next data for writing..>
            // Send next packet of data.
            r = USBD_VSC_Write(_hInst, &ac[0], 200, -1);
            if (r < 0) {
              <.. error handling..>
            }
   }
}
// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBD_VSC_SetOnEPEvent(hInst, &_usb_callback, _OnEvent, NULL);
// Send the first packet of data using an asynchronous write operation.
r = USBD_VSC_Write(_hInst, &ac[0], 200, -1, 0);
if (r < 0) {
  <.. error handling..>
}
<.. do anything else here while the whole data is sent..>
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.14 USBD_VSC_TxIsPending()

Description

Checks whether the TX (IN endpoint) is currently pending. Can be called in any context.

Prototype

```
int USBD_VSC_TxIsPending(U8 EPIndex);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().

Return value

1 We have queued data to be sent.
0 Queue is empty.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.15 USBD_VSC_WaitEP()

Description

Waits (blocking) until the triggered USBD_VSC_ReadOverlapped() has received the desired data.

Prototype

```
int USBD_VSC_WaitEP(U8       EPIndex,
                    unsigned Timeout);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
= 1 Timeout occurred.
< 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

Example

```
if (USBD_VSC_Read(hInst, &ac[0], 50, -1, 0) < 0) {
  <.. error handling..>
  return;
}
//
// USBD_VSC_Read() with Timeout==-1 will return immediately.
// Do something else while data may be transferred.
//
<...>
//
// Now wait until we get all 50 bytes.
// USBD_VSC_WaitEP() will block, until a total of
// 50 bytes are read or timeout occurs.
//
if (USBD_VSC_WaitEP(hInst, timeout) != 0) {
  <.. timeout error handling..>
  return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.16 USBD_VSC_PollEP()

Description

Waits (blocking) until the triggered USBD_VSC_ReadOverlapped() has received the desired data.

Prototype

```
int USBD_VSC_PollEP(U8       EPIndex,
                    unsigned Timeout);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

= 0 Transfer completed.
= 1 Timeout occurred.
< 0 An error occurred (e.g. target disconnected)

Additional information

In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example for an IN EP

```
if (USBD_VSC_Write(hInst, &ac[0], 50, -1, 0) < 0) {
  <.. error handling..>
  return;
}
//
// USBD_VSC_Write() will return immediately.
// While waiting for the data to be transferred, we will blink a LED with
 200 ms interval.
// USBD_VSC_PollForTX() will return, if all data were sent or 100 ms expired.
//
while ((r = USBD_VSC_PollEP(hInst, 100)) > 0) {
  ToggleLED();
}
if (r < 0) {
  <.. error handling..>
  return;
}
// Now all data have been sent.
```

Example for an OUT EP

```
if (USBD_VSC_Read(hInst, &ac[0], 50, -1, 0) < 0) {
  <.. error handling..>
  return;
}
//
// USBD_VSC_Read() with Timeout==-1 will return immediately.
// While waiting for the data, we will blink a LED with 200 ms interval.
// USBD_VSC_PollForRX() will return if all data were read or 100 ms expired.
//
while ((r = USBD_VSC_PollEP(hInst, 100)) > 0) {
  ToggleLED();
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
if (r < 0) {
  <.. error handling..>
  return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.17 USBD_VSC_WaitForTXReady()

Description

Waits (blocking) until the TX queue can accept another data packet. This function is used in combination with a non-blocking call to USBD_VSC_Write() , it waits until a new asynchronous write data transfer will be accepted by the USB stack.

Prototype

```
int USBD_VSC_WaitForTXReady(U8  EPIndex,
                            int Timeout);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is negative, the function will return immediately.

Return value

= 0	A new asynchronous write data transfer will be accepted.
= 1	The write queue is full, a call to USBD_VSC_Write() would return USB_STATUS_EP_BUSY.
< 0	Error occurred.

Additional information

If Timeout is 0, the function never returns 1.

If Timeout is -1, the function will not wait, but immediately return the current state.

Example

```
// Always keep the write queue full for maximum send speed.
for (;;) {
  pData = GetNextData(&NumBytes);
  // Wait until stack can accept a new write.
  USBD_VSC_WaitForTxReady(hInst, 0);
  // Issue write transfer.
  if (USBD_VSC_Write(hInst, pData, NumBytes, -1) < 0) {
    <.. error handling..>
  }
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.18 USBD_VSC_Write()

Description

Sends data to the USB host. Depending on the Timeout parameter, the function blocks until NumBytes have been written or a timeout occurs.

Prototype

```
int USBD_VSC_Write(      U8          EPIndex,
                   const void      * pData,
                         unsigned    NumBytes,
                         int         Timeout,
                         unsigned    Flags);
```

Parameters

Parameter	Description
EPIndex	One of the EPIndex was used in pInitData when calling USBD_VSC_Add().
pData	Data that should be written.
NumBytes	Number of bytes to write.
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.
Flags	Various flags: Current flags: • USB_VSC_WRITE_FLAG_NO_NULL_PACKET - Specifies that a zero-length packet shall be sent when the last data packet is a multiple of MaxPacketSize. Normally MaxPacketSize for full-speed devices is 64 bytes. For high-speed devices the normal packet size is between 64 and 512 bytes.

Return value

= 0	Successfully started an asynchronous write transfer or a timeout has occurred and no data was written.
> 0 && < NumBytes	Number of bytes that have been written before a timeout occurred.
= NumBytes	Write transfer successfully completed.
< 0	Error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

The USB stack is able to queue a small number of asynchronous write transfers (when using Timeout = -1). If a write transfer is still in progress when this function is called and the USB stack can not accept another write transfer request, the functions returns USB_STATUS_EP_BUSY. A synchronous write transfer (Timeout ≥ 0) will always block until the transfer (including all pending transfers) are finished or a timeout occurs.

In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 136).

In order to synchronize, USBD_VSC_WaitForTX() needs to be called. Another synchronization method would be to periodically call USBD_VSC_GetNumBytesRemToWrite() in order to see how many bytes still need to be written (this method is preferred when a non-blocking solution is necessary). The write operation can be canceled using USBD_VSC_CancelWrite().

If pData = NULL and NumBytes = 0, a zero-length packet is sent to the host.

 © 2010-2026 SEGGER Microcontroller GmbH

The content of the buffer pointed to by `pData` must not be changed until the transfer has been completed.

Example

```
NumBytesWritten = USBD_VSC_Write(hInst, &ac[0], DataSize, 500);
if (NumBytesWritten <= 0) {
  <.. error handling..>
}
if (NumBytesWritten < DataSize) {
  <.. timeout occurred, data partially written within 500ms ..>
} else {
  <.. write completed successfully..>
}
```

See also *USBD_VSC_GetNumBytesRemToWrite* on page 225.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.2.19 USBD_VSC_WriteAsync()

Description

Sends data to the host asynchronously. The function does not block. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_VSC_WriteAsync(U8                       EPIndex,
                         USB_ASYNC_IO_CONTEXT * pContext,
                         I8                       Send0PacketIfRequired);
```

Parameters

Parameter	Description
EPIndex	A valid EP Index that was also passed to USBD_VSC_Add().
pContext	Pointer to a structure of type USB_ASYNC_IO_CONTEXT containing parameters and a pointer to the callback function.
Send0PacketIfRequired	Specifies that a zero-length packet shall be sent when the last data packet is a multiple of MaxPacketSize.

Example

```
static void _AsyncCb(USB_ASYNC_IO_CONTEXT * pIOContext) {
  U8  *p;

  p = (U8 *)pIOContext->pContext;
  *p = 1;
}

<...>

USB_ASYNC_IO_CONTEXT IOContext;
U8                   AsyncComplete;

IOContext.NumBytesToTransfer = 5000;
IOContext.pData              = pBuff;
IOContext.pfOnComplete       = _AsyncCb;
IOContext.pContext           = (void *)&AsyncComplete;
AsyncComplete = 0;
USBD_VSC_WriteAsync(hInst, &IOContext, 1);
while (AsyncComplete == 0) {
  <.. Do other work. ..>
}
// Transaction is complete.
if (IOContext.Status < 0 || IOContext.NumBytesTransferred != 5000) {
  <.. error handling ..>
} else {
  <.. data written successfully ..>
}
<...>
```

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.3 Data structures

6.4.3.1 USB_VSC_INIT_DATA

Description

Initialization structure that is needed when adding a VSC interface to emUSB-Device.

Type definition

```
typedef struct {
  U16                            Flags;
  U8                             aEP[USBD_VSC_MAX_EPS];
  U8                             NumEPs;
  const char                   * pInterfaceName;
  U8                             InterfaceClass;
  U8                             InterfaceSubClass;
  U8                             InterfaceProtocol;
  const USB_VSC_MSOSDESC_INFO  * pMSDescInfo;
} USB_VSC_INIT_DATA;
```

Structure members

Member	Description
Flags	Various flags. Currently only one flag is available: • USB_VSC_USE_CUSTOM_MSOSDESC: Allows to use a custom specified MS OS Descriptor. Otherwise this has to be be initialized to 0.
aEP	Array of Endpoints Indices to be used. Each EPIndex needs to be allocated by USBD_AddEP
NumEPs	Number of EPIndex in array.
pInterfaceName	Name of the interface.
InterfaceClass	Sets the USB Class ID .
InterfaceSubClass	Sets the USB SubClass ID.
InterfaceProtocol	Sets the USB Protocol ID.
pMSDescInfo	[Optional] This pointer will only be used when the Flag USB_VSC_USE_CUSTOM_MSOSDESC is set.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.3.2 USB_VSC_MSOSDESC_INFO

Description

MS OS descriptor structure that contains for MS related OSes information how to deal with device with out having a driver store.

Type definition

```
typedef struct {
  const char                * sCompatibleID;
  const char                * sSubCompatibleID;
  U32                         NumProperties;
  const USB_MS_OS_EXT_PROP  * pProperties;
} USB_VSC_MSOSDESC_INFO;
```

Structure members

Member	Description
sCompatibleID	Strings that gives MS OS a hint to the driver that shall be loaded
sSubCompatibleID	[Optional] Gives a sub id string, in most cases this can be NULL.
NumProperties	NumBer of properties that are stored in pProperties
pProperties	Variable array of MS OS extended OS descriptors. Depending on the sCompatibleID, this can be some sub structure which will be stored in the Windows registry. Eg. with WinUSB this contains the GUID which is needed to identify your device among other WinUSB devices.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.3.3 USB_VSC_ON_ADD_FUNCTION_DESC

Description

Call back that is used to add an additional descriptor between the interface or one of its alternate setting descriptor and the endpoint descriptor(s).

Type definition

```
typedef const U8 * (USB_VSC_ON_ADD_FUNCTION_DESC)
                                          (USB_VSC_HANDLE hInst,
                                           U8             IFAlternateSetting);
```

Parameters

Parameter	Description
hInst	Handle to a valid VSC instance, returned by USBD_VSC_Add().
IFAlternateSetting	Data that should be written.

Return value

= NULL No additional descriptor shall be added to this interface or its alternate setting..
≠ NULL Pointer to a USB descriptor.

Additional information

USB Descriptor follow a specific format. The first byte is always the length. The second byte describes the descriptor type. Anything after these 2 bytes is descriptor dependent.

 © 2010-2026 SEGGER Microcontroller GmbH

6.4.3.4 USB_VSC_ON_SET_INTERFACE

Description

Global callback function that is called whenever an alternate setting is set for an interface that was added with USBD_VSC_Add().

Type definition

```
typedef void (USB_VSC_ON_SET_INTERFACE)(USB_VSC_HANDLE hInst,
                                        U8             AlternateInterface);
```

Parameters

Parameter	Description
hInst	Handle to a valid VSC instance, returned by USBD_VSC_Add().
AlternateInterface	Alternate interface that was set by the host.

Additional information

Each interface has one alternate setting which is the default setting. This call back is called after the host has set the alternate setting. By default all data transfers of the previous interface/alternate setting are canceled.

 © 2010-2026 SEGGER Microcontroller GmbH

6.5 C++ Host API

This chapter describes the functions that can be used with the host system.

To communicate with a target running the USB VSC or BULK class, an API is provided that can be used on Windows, Linux, and macOS systems.

To have an easy start-up when writing an application on the host side, these API functions have a simple interface and handle all required operations to communicate with the target VSC class.

To use the API in an application, the header file `USBVAPI.h` must be included. Depending on the host operating system used, one of the following libraries must be added to the host application:

- Windows: `USBVAPI_r_Windows_x86_64.a`.
- Linux: `USBVAPI_r_Linux_x86_64.a`.

The documentation for the host API is provided as HTML in the file `USBVAPI_documentation_html.zip`. To view the documentation, unzip this file and open the `index.html` from the resulting (unzipped) directory in a web browser.

The following table shows where components of the C++ API can be found in the emUSB-Device shipping:

File / Directory	Contents
`Windows/USB/BULK/Inc/USBVAPI.h`	Header file for the API
`Doc/USBVAPI_documentation_html.zip`	Manual of the API.
`Windows/USB/BULK/USBVAPI`	Libraries for different host architectures
`Windows/USB/BULK/USBVAPI_SampleApplication`	Sample applications to demonstrate the usage of the API.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 7

Mass Storage Device Class (MSD)

This chapter gives a general overview of the MSD class and describes how to get the MSD component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

7.1 Overview

The Mass Storage Device (MSD) is a USB class protocol defined by the USB Implementers Forum. The class itself is used to access one or more storage devices such as flash drives or memory sticks.

As the USB mass storage device class is well standardized, every major operating system such as Microsoft Windows (after Windows 2000), Apple OS X, Linux and many more support it. Therefore an installation of a custom host USB driver is normally not necessary.

emUSB-Device-MSD comes as a whole packet and contains the following:

- Generic USB handling
- MSD device class implementation, including support for direct disk and CD-ROM mode (CD-ROM access is a separate component)
- Several storage drivers for handling different devices
- Example applications

 © 2010-2026 SEGGER Microcontroller GmbH

7.2 MSD Configuration

7.2.1 Initial configuration

To get emUSB-Device-MSD up and running and to do an initial test, the configuration as it is delivered should not be modified.

7.2.2 Final configuration

The configuration must only be modified when emUSB-Device is deployed in your final product. Refer to *emUSB-Device Configuration* on page 53 for detailed information about the generic information functions which must be adapted.

In order to comply with the Mass Storage Device Bootability specification, the serial number provided by the function `USBD_SetDeviceInfo()` must be a string with at least 12 characters, where each character is a hexadecimal digit (‘0’ through ‘9’ or ‘A’ through ‘F’).

7.2.3 MSD class specific configuration functions

Besides the generic emUSB-Device configuration functions (*emUSB-Device Configuration* on page 53), the following should be adapted before the emUSB-Device MSD component is used in a final product. Example implementations are supplied in the MSD example application `USB_MSD_FS_Start.c`, located in the `Application` directory of emUSB-Device.

Each logical unit (storage) which is added to the MSD component has its own set of name and ID values which is supplied when the logical unit is first added through `USBD_MSD_AddUnit()`

Example

```
static const USB_MSD_LUN_INFO _Lun0Info = {
  "Vendor",     // MSD VendorName
  "MSD Volume", // MSD ProductName
  "1.00",       // MSD ProductVer
  "134657890"   // MSD SerialNo
};
...
InstData.pLunInfo = &_Lun0Info;
...
USB_MSD_AddUnit(&InstData);
```

7.2.4 Running the example application

The directory `Application` contains example applications that can be used with emUSB-Device and the MSD component. To test the emUSB-Device-MSD component, build and download the application of choice into the target. Remove the USB connection and reconnect the target to the host. The target will enumerate and can be accessed via a file browser.

 © 2010-2026 SEGGER Microcontroller GmbH

7.2.4.1 MSD_Start_StorageRAM.c in detail

The main part of the example application USB_MSD_Start_StorageRAM.c is implemented in a single task called MainTask().

```
/* MainTask() - excerpt from USB_MSD_Start_StorageRAM.c */
void MainTask(void);
void MainTask(void) {
  USBD_Init();
  _AddMSD();
  USBD_Start();
  while (1) {
    while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
            != USB_STAT_CONFIGURED) {
      BSP_ToggleLED(0);
      USB_OS_Delay(50);
    }
    BSP_SetLED(0);
    USBD_MSD_Task();
  }
}
```

The first step is to initialize the USB core stack using USBD_Init(). The function _AddMSD() configures all required endpoints and assigns the used storage medium to the MSD component.

```
/* _AddMSD() - excerpt from MSD_Start_StorageRAM.c */
static void _AddMSD(void) {
  static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
  USB_MSD_INIT_DATA     InitData;
  USB_MSD_INST_DATA     InstData;
  InitData.EPIn  = USBD_AddEP(1, USB_TRANSFER_TYPE_BULK,
                              USB_HS_BULK_MAX_PACKET_SIZE, NULL, 0);
  InitData.EPOut = USBD_AddEP(0, USB_TRANSFER_TYPE_BULK,
                              USB_HS_BULK_MAX_PACKET_SIZE,
                              _abOutBuffer, sizeof(_abOutBuffer));
  USBD_MSD_Add(&InitData);
  //
  // Add logical unit 0: RAM drive
  //
  memset(&InstData, 0,  sizeof(InstData));
  InstData.pAPI                    = &USB_MSD_StorageRAM;
  InstData.DriverData.pStart       = (void*)MSD_RAM_ADDR;
  InstData.DriverData.NumSectors   = MSD_RAM_NUM_SECTORS;
  InstData.DriverData.SectorSize   = MSD_RAM_SECTOR_SIZE;
  InstData.pLunInfo = &_Lun0Info;
  USBD_MSD_AddUnit(&InstData);
}
```

The example application uses a RAM disk as a storage medium.

The example RAM disk has a size of 23 kB (46 sectors with a sector size of 512 bytes). You can increase the size of the RAM disk by modifying the macros MSD_RAM_NUM_SECTORS and MSD_RAM_SECTOR_SIZE (in multiples of 512), but the size must be at least 23 kB, otherwise a Windows host cannot format the disk.

```
/* AddMSD() - excerpt from MSD_Start_StorageRAM.c */
#define MSD_RAM_NUM_SECTORS  46
#define MSD_RAM_SECTOR_SIZE  512
```

 © 2010-2026 SEGGER Microcontroller GmbH

7.3 Target API

Function	Description
API functions	
USBD_MSD_Add()	Adds an MSD-class interface to the USB stack.
USBD_MSD_AddUnit()	Adds a mass storage device to emUSB-Device-MSD.
USBD_MSD_AddCDRom()	Adds a CD-ROM device to emUSB-Device-MSD.
USBD_MSD_SetPreventAllowRemovalHook()	Sets a callback function to prevent/allow removal of storage medium.
USBD_MSD_SetReadWriteHook()	Sets a callback function which gives information about the read and write block-wise operations to the storage medium.
USBD_MSD_Task()	Task that handles the MSD-specific protocol.
USBD_MSD_Poll()	Function which handles MSD commands.
USBD_MSD_PollEx()	Function which handles MSD commands.
USBD_MSD_SetStartStopUnitHook()	Sets a callback function which is called when the command StartStopUnit is called.
Extended API functions	
USBD_MSD_Connect()	Connects the storage medium to the MSD component.
USBD_MSD_Disconnect()	Disconnects the storage medium from the MSD.
USBD_MSD_RequestDisconnect()	Sets the DisconnectRequest flag.
USBD_MSD_RequestRefresh()	Performs a disconnect (optional), a detach and optionally a re-attach, to inform host that volume contents has changed.
USBD_MSD_UpdateWriteProtect()	This function updates the write protect status of the storage medium.
USBD_MSD_WaitForDisconnection()	Waits for disconnection while time out is not reached.
Data structures	
USB_MSD_INIT_DATA	emUSB-Device-MSD initialization structure that is required when adding an MSD interface.
USB_MSD_INFO	emUSB-Device-MSD storage interface.
USB_MSD_INST_DATA	USB-MSD initialization structure that is required when adding an MSD interface.
PREVENT_ALLOW_REMOVAL_HOOK	Callback function to prevent/allow removal of storage medium.

 © 2010-2026 SEGGER Microcontroller GmbH

Function	Description
READ_WRITE_HOOK	Callback function which is called with every read/write access to the storage medium.
USB_MSD_INST_DATA_DRIVER	USB-MSD initialization structure that is required when adding an MSD interface.
USB_MSD_STORAGE_API	USB-MSD initialization structure that is required when adding an MSD interface.
START_STOP_UNIT_HOOK	Callback function which is called when a START STOP UNIT SCSI command is received.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.1 API functions

7.3.1.1 USBD_MSD_Add()

Description

Adds an MSD-class interface to the USB stack.

Prototype

```
void USBD_MSD_Add(const USB_MSD_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a USB_MSD_INIT_DATA structure.

Additional information

After the initialization of general emUSB-Device, this is the first function that needs to be called when an MSD interface is used with emUSB-Device. The structure USB_MSD_INIT_DATA must be initialized before USBD_MSD_Add() is called.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.1.2 USBD_MSD_AddUnit()

Description

Adds a mass storage device to emUSB-Device-MSD.

Prototype

```
void USBD_MSD_AddUnit(const USB_MSD_INST_DATA * pInstData);
```

Parameters

Parameter	Description
pInstData	Pointer to a USB_MSD_INST_DATA structure containing the information of the added storage device.

Additional information

It is necessary to call this function immediately after USBD_MSD_Add(). It will then add an R/W storage device to emUSB-Device-MSD. The structure USB_MSD_INST_DATA must be initialized before calling USBD_MSD_AddUnit().

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.1.3 USBD_MSD_AddCDRom()

Description

Adds a CD-ROM device to emUSB-Device-MSD.

Prototype

```
void USBD_MSD_AddCDRom(const USB_MSD_INST_DATA * pInstData);
```

Parameters

Parameter	Description
pInstData	Pointer to a USB_MSD_INST_DATA structure containing the information of the added storage device.

Additional information

Similar to USBD_MSD_AddUnit(), this function should be called after USBD_MSD_Add(). The structure USB_MSD_INST_DATA must be initialized before USBD_MSD_AddCDRom() is called.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.1.4 USBD_MSD_SetPreventAllowRemovalHook()

Description

Sets a callback function to prevent/allow removal of storage medium.

Prototype

```
void USBD_MSD_SetPreventAllowRemovalHook
                              (U8                          Lun,
                               PREVENT_ALLOW_REMOVAL_HOOK * pfOnPreventAllowRemoval);
```

Parameters

Parameter	Description
Lun	Logical Unit Number. Using only one storage medium, this parameter is 0.
pfOnPreventAllowRemoval	Pointer to the callback function that shall be called.

Additional information

The callback is called within the MSD task context. The callback must not block.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.1.5 USBD_MSD_SetReadWriteHook()

Description

Sets a callback function which gives information about the read and write block-wise operations to the storage medium.

Prototype

```
void USBD_MSD_SetReadWriteHook(U8                Lun,
                               READ_WRITE_HOOK * pfOnReadWrite);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.
pfOnReadWrite	Pointer to the callback function that shall be called.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.1.6 USBD_MSD_Task()

Description

Task that handles the MSD-specific protocol.

Prototype

```
void USBD_MSD_Task(void);
```

Additional information

After the USB device has been successfully enumerated and configured, the `USBD_MSD_Task()` should be called. This function blocks until the device is detached or is suspended. After a detach or suspend `USBD_MSD_Task()` will return.

Check `USBD_MSD_Poll()` if you need a non-blocking version.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.1.7 USBD_MSD_Poll()

Description

Function which handles MSD commands. Using this function is only necessary if you want to avoid using the blocking `USB_MSD_Task` function. This can be necessary if you are not using an RTOS.

Prototype

```
int USBD_MSD_Poll(void);
```

Return value

2	O.K. Command was processed, but a protocol error occurred.
1	O.K. Command was processed successfully.
0	O.K. Timeout occurred.
	1 An error occurred. (e.g. no cable connected).

Additional information

This function must be called periodically, otherwise the host can time out the device since it does not process commands. It normally blocks for `USB_MSD_POLL_TIMEOUT` milliseconds while waiting for a command from the host. Should a command arrive during the timeout it will be processed, which could potentially increase the block duration. The duration could also decrease because the function returns as soon as a command is finished.

In case of return value 2 the calling task should still call this function again, if possible recovery will be initiated.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.1.8 USBD_MSD_PollEx()

Description

Function which handles MSD commands. Using this function is only necessary if you want to avoid using the blocking `USB_MSD_Task` function. This can be necessary if you are not using an RTOS.

Prototype

```
int USBD_MSD_PollEx(unsigned Timeout);
```

Parameters

Parameter	Description
`Timeout`	Function will block for 'Timeout' ms, if no requests are received from the host. `Timeout` may be zero for non-blocking mode.

Return value

2 O.K. Command was processed, but a protocol error occurred.
1 O.K. Command was processed successfully.
0 O.K. `Timeout` occurred.
1 An error occurred. (e.g. no cable connected).

Additional information

This function must be called periodically, otherwise the host can time out the device since it does not process commands. It normally blocks for 'Timeout' milliseconds while waiting for a command from the host. Should a command arrive during the timeout it will be processed, which could potentially increase the block duration. The duration could also decrease because the function returns as soon as a command is finished.

In case of return value 2 the calling task should still call this function again, if possible recovery will be initiated.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.1.9 USBD_MSD_SetStartStopUnitHook()

Description

Sets a callback function which is called when the command StartStopUnit is called.

Prototype

```
void USBD_MSD_SetStartStopUnitHook(U8                      Lun,
                                   START_STOP_UNIT_HOOK * pfOnStartStopUnit);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.
pfOnStartStopUnit	Pointer to the callback function that shall be called. For detailed information about the function pointer, refer to START_STOP_UNIT_HOOK.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.2 Extended API functions

7.3.2.1 USBD_MSD_Connect()

Description

Connects the storage medium to the MSD component.

Prototype

```
void USBD_MSD_Connect(U8 Lun);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.

Additional information

The storage medium is initially always connected to the MSD component. This function is normally used after the storage medium was disconnected via `USBD_MSD_Disconnect()` to carry out file system operations on the device application side. Because the device can not actively perform a connect operation this function sets an internal flag and the next time when the host requests the status of the storage medium the storage medium is connected back to the MSD component.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.2.2 USBD_MSD_Disconnect()

Description

Disconnects the storage medium from the MSD.

Prototype

```
void USBD_MSD_Disconnect(U8 Lun);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.

Additional information

This function will force the storage medium to be disconnected. The host will be informed that the medium is not present. In order to reconnect the device to the host, the function `USBD_MSD_Connect()` shall be used. See `USBD_MSD_RequestDisconnect()` and `USBD_MSD_WaitForDisconnection()` for a graceful disconnection method.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.2.3 USBD_MSD_RequestDisconnect()

Description

Sets the DisconnectRequest flag.

Prototype

```
void USBD_MSD_RequestDisconnect(U8 Lun);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.

Additional information

This function sets the disconnect flag for the storage medium. As soon as the next MSD command is sent to the device, the host will be informed that the device is currently not available. To reconnect the storage medium, `USBD_MSD_Connect()` shall be called.

Notes

If the host tries to access the storage medium while this flag is set to 1, the status of the storage medium changes to disconnected.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.2.4 USBD_MSD_RequestRefresh()

Description

Performs a disconnect (optional), a detach and optionally a re-attach, to inform host that volume contents has changed.

Prototype

```
void USBD_MSD_RequestRefresh(U8  Lun,
                             U32 Flags);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.
Flags	Request flags, a bit-ored combination of the following flags: • USB_MSD_TRY_DISCONNECT - Try a medium disconnect before doing a USB detach. • USB_MSD_RE_ATTACH - Automatically re-attach after detach has been done.

Additional information

If the flag USB_MSD_TRY_DISCONNECT is given, the function sets the disconnect flag for the storage medium. As soon as the next MSD command is sent to the device, the host will be informed that the device is currently not available. If the host acknowledges the disconnect, the medium is reconnected and the function USBD_MSD_Task() will return.

If the flag USB_MSD_TRY_DISCONNECT is not set or the host ignores the disconnection of the medium, the USB device is detached from the host (using USBD_Stop()).

If the flag USB_MSD_RE_ATTACH is set, the device is re-attached after some delay using USBD_Start(). Then the function USBD_MSD_Task() will return. The function USBD_MSD_RequestRefresh() returns immediately while the procedure is executed in the USBD_MSD_Task().

Returning of the function USBD_MSD_Task() allows the application to reinitialize the volume (or calling USBD_Start(), if USB_MSD_RE_ATTACH was not set) before calling USBD_MSD_Task() again.

Detaching the USB device not only affects the specified volume (Lun) but all volumes of the device and any other USB class interfaces.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.2.5 USBD_MSD_UpdateWriteProtect()

Description

This function updates the write protect status of the storage medium.

Prototype

```
void USBD_MSD_UpdateWriteProtect(U8 Lun,
                                 U8 IsWriteProtected);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.
IsWriteProtected	Set the write protect flag: 1 - Medium is write-protected. 0 - Medium is NOT write-protected.

Additional information

Please make sure that this function is called when the LUN is disconnected from the host, otherwise the change in the WriteProtected flag is normally not recognized.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.2.6 USBD_MSD_WaitForDisconnection()

Description

Waits for disconnection while time out is not reached.

Prototype

```
int USBD_MSD_WaitForDisconnection(U8  Lun,
                                  U32 TimeOut);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one storage medium, this parameter is 0.
TimeOut	Timeout given in ms. How long should this function wait, until it stops waiting.

Return value

0 Error - Time out reached. Device not disconnected.
1 Success - Device disconnected.

Additional information

After triggering the disconnection via `USBD_MSD_RequestDisconnect()` the stack disconnects the storage medium as soon as the host requests the status of the storage medium. Win2k does not periodically check the status of a USB MSD. Therefore, the timeout is required to leave the loop. The return value can be used to decide if the disconnection should be forced. In this case, `USBD_MSD_Disconnect()` shall be called.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.3 Data structures

7.3.3.1 USB_MSD_INIT_DATA

Description

emUSB-Device-MSD initialization structure that is required when adding an MSD interface.

Type definition

```
typedef struct {
  U8   EPIn;
  U8   EPOut;
  U8   InterfaceNum;
} USB_MSD_INIT_DATA;
```

Structure members

Member	Description
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host.
InterfaceNum	Interface number. This member is used internally, set to 0.

Additional information

This structure holds the endpoints that should be used with the MSD interface. Refer to USBD_AddEP() for more information about how to add an endpoint.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.3.2 USB_MSD_INFO

Description

emUSB-Device-MSD storage interface.

Type definition

```
typedef struct {
  U32   NumSectors;
  U16   SectorSize;
} USB_MSD_INFO;
```

Structure members

Member	Description
NumSectors	Number of available sectors.
SectorSize	Size of one sector in bytes.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.3.3 USB_MSD_INST_DATA

Description

USB-MSD initialization structure that is required when adding an MSD interface.

Type definition

```
typedef struct {
  const USB_MSD_STORAGE_API   * pAPI;
  USB_MSD_INST_DATA_DRIVER      DriverData;
  U8                            DeviceType;
  U8                            IsPresent;
  USB_MSD_HANDLE_CMD          * pfHandleCmd;
  U8                            IsWriteProtected;
  const USB_MSD_LUN_INFO      * pLunInfo;
} USB_MSD_INST_DATA;
```

Structure members

Member	Description
pAPI	Pointer to a structure that holds the storage device driver API.
DriverData	Driver data that are passed to the storage driver. Refer to USB_MSD_INST_DATA_DRIVER for detailed information about how to initialize this structure.
DeviceType	Determines the type of the device: 0: Direct access block device 5: CD/DVD
IsPresent	Determines if the medium is storage is present. For non-removable devices always 1.
pfHandleCmd	Optional pointer to a callback function which handles SCSI commands.
IsWriteProtected	Specifies whether the storage medium shall be write-protected.
pLunInfo	Pointer to a USB_MSD_LUN_INFO structure. Filling this structure is mandatory for each LUN.

Additional information

All non-optional members of this structure need to be initialized correctly, except DeviceType and pfHandleCmd because it is done by the functions USBD_MSD_AddUnit() or USBD_MSD_AddCDROM().

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.3.4 USB_MSD_LUN_INFO

Description

Structure that is used when adding a logical volume to emUSB-Device-MSD.

Type definition

```
typedef struct {
  const char  * pVendorName;
  const char  * pProductName;
  const char  * pProductVer;
  const char  * pSerialNo;
} USB_MSD_LUN_INFO;
```

Structure members

Member	Description
pVendorName	Vendor name of the mass storage device. The string should be no longer than 8 bytes.
pProductName	Product name of the mass storage device. The product name string should be no longer than 16 bytes.
pProductVer	Product version number of the mass storage device. The product version string should be no longer than 4 bytes.
pSerialNo	Product serial number of the mass storage device. The serial number string must be exactly 12 bytes, in order to satisfy the USB bootability specification requirements.

Additional information

The setting of these values is mandatory. If these values remain NULL at initialization, emUSB-Device will report a panic error in debug builds (USB_PANIC).

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.3.5 PREVENT_ALLOW_REMOVAL_HOOK

Description

Callback function to prevent/allow removal of storage medium. See `USBD_MSD_SetPreventAllowRemovalHook()`.

Type definition

```
typedef void (PREVENT_ALLOW_REMOVAL_HOOK)(U8 PreventRemoval);
```

Parameters

Parameter	Description
`PreventRemoval`	Show whether the device shall be locked or not. • 0 - The device shall be removable. • 1 - The device shall be locked.

Additional information

Most OSes call the prevent/allow removal before any write operation. This callback will be called for all LUNs that are available on the host.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.3.6 READ_WRITE_HOOK

Description

Callback function which is called with every read/write access to the storage medium.

Type definition

```
typedef void (READ_WRITE_HOOK)(U8  Lun,
                               U8  IsRead,
                               U8  OnOff,
                               U32 StartLBA,
                               U32 NumBlocks);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number which was accessed through read or write.
IsRead	Specifies whether a read or a write access was used: • 1 : read • 0 : write
OnOff	States whether the read or write request has been initialized (1) or whether it is complete (0).
StartLBA	The first Logical Block Address accessed by the transfer.
NumBlocks	The number of blocks accessed by the transfer, starting from the StartLBA.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.3.7 USB_MSD_INST_DATA_DRIVER

Description

USB-MSD initialization structure that is required when adding an MSD interface.

Type definition

```
typedef struct {
  void     * pStart;
  U32        StartSector;
  U32        NumSectors;
  U16        SectorSize;
  void     * pSectorBuffer;
  unsigned   NumBytes4Buffer;
  U8         NumBuffers;
} USB_MSD_INST_DATA_DRIVER;
```

Structure members

Member	Description
pStart	A pointer defining the start address
StartSector	The start sector that is used for the driver.
NumSectors	The available number of sectors available for the driver.
SectorSize	The sector size that should be used by the driver.
pSectorBuffer	Pointer to an application provided buffer to be used as temporary buffer for storing the sector data.
NumBytes4Buffer	Size of the application provided buffer.
NumBuffers	Number of buffer that are available. This is only used when using the MT storage layer.

Additional information

This structure is passed to the storage driver. Therefore, the members of this structure can depend on the driver that is used. For the storage drivers that are shipped with this software, the members of USB_MSD_INST_DATA_DRIVER have the following meaning:

USB_MSD_StorageRAM:

Member	Description
pStart	A pointer defining the start address of the RAM disk.
StartSector	This member is ignored.
NumSectors	The number of sectors available for the RAM disk.
SectorSize	The sector size that should be used by the driver.

USB_MSD_StorageByName:

Member	Description
pStart	Pointer to a string holding the name of the volumes that shall be used, for example "nand:" "mmc:1:"
StartSector	Specifies the start sector.
NumSectors	Number of sectors that shall be used.
SectorSize	This member is ignored.
pSectorBuffer	Pointer to an application-provided buffer to be used as a temporary buffer for storing the sector data.
NumBytes4Buffer	Size of the buffer provided by the application. Please make sure that the buffer can hold at least 3 sectors; otherwise,

 © 2010-2026 SEGGER Microcontroller GmbH

Member	Description
	`pSectorBuffer` and `NumBytes4Buffer` are ignored and an internal sector buffer is used. This sector buffer is then allocated by using the FS-Storage-Layer functions.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.3.8 USB_MSD_STORAGE_API

Description

USB-MSD initialization structure that is required when adding an MSD interface.

Type definition

```
typedef struct {
  USB_MSD_STORAGE_INIT             * pfInit;
  USB_MSD_STORAGE_GETINFO          * pfGetInfo;
  USB_MSD_STORAGE_GETREADBUFFER    * pfGetReadBuffer;
  USB_MSD_STORAGE_READ             * pfRead;
  USB_MSD_STORAGE_GETWRITEBUFFER   * pfGetWriteBuffer;
  USB_MSD_STORAGE_WRITE            * pfWrite;
  USB_MSD_STORAGE_MEDIUMISPRESENT  * pfMediumIsPresent;
  USB_MSD_STORAGE_DEINIT           * pfDeInit;
} USB_MSD_STORAGE_API;
```

Structure members

Member	Description
pfInit	Initializes the storage medium.
pfGetInfo	Retrieves storage medium information such as sector size and number of sectors available.
pfGetReadBuffer	Prepares read function and returns a pointer to a buffer that is used by the storage driver.
pfRead	Reads one or multiple sectors from the storage medium.
pfGetWriteBuffer	Prepares write function and returns a pointer to a buffer that is used by the storage driver.
pfWrite	Writes one or more sectors to the storage medium.
pfMediumIsPresent	Checks if medium is present.
pfDeInit	De-initializes the storage medium.

Additional information

USB_MSD_STORAGE_API is used to retrieve information from the storage device driver or access data that needs to be read or written. Detailed information can be found in *MSD Storage Driver* on page 281.

 © 2010-2026 SEGGER Microcontroller GmbH

7.3.3.9 START_STOP_UNIT_HOOK

Description

Callback function which is called when a START STOP UNIT SCSI command is received.

Type definition

```
typedef void (START_STOP_UNIT_HOOK)(U8 Lun,
                                    U8 StartLoadEject);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
StartLoadEject	Specifies which operation is executed by the host: • 0 : Stop disk • 1 : Start disk and make ready for access • 2 : Eject disk if permitted • 3 : Load, start and make disk ready.

 © 2010-2026 SEGGER Microcontroller GmbH

7.4 MSD Storage Driver

7.4.1 General information

The storage interface is handled through an API-table, which contains all relevant functions necessary for read/write operations and initialization. Its implementation handles the details of how data is actually read from or written to memory. Additionally, MSD knows two different media types:

- Direct media access, for example RAM-Disk, NAND flash, MMC/SD cards etc.
- CD-ROM emulation.

7.4.1.1 Supported storage types

The supported storage types include:

- RAM, directly connected to the processor via the address bus.
- External flash memory, e.g., SD cards.
- Mechanical drives, for example, CD-ROM. This is essentially an ATA/SCSI to USB bridge.

7.4.1.2 Storage drivers supplied with this release

The following drivers are included:

- `USB_MSD_StorageRAM`: A RAM driver which should work with almost any device.
- `USB_MSD_StorageByIndex`: A storage driver that uses the storage layer (logical block layer) of emFile to access the device.
- `USB_MSD_StorageByName`: A storage driver that uses the storage layer (logical block layer) of emFile to access the device.

> **Note**
>
> If you are not using emFile or the RAM driver, you will have to provide your own sector write/read routines for your storage medium.

7.4.2 Interface function list

As described above, access to a storage medium is realized through an API-function table (`USB_MSD_STORAGE_API`). The storage functions are declared in `USB_MSD.h`.

 © 2010-2026 SEGGER Microcontroller GmbH

7.4.3 USB_MSD_STORAGE_API in detail

7.4.3.1 USB_MSD_STORAGE_INIT

Description

Initializes the storage medium.

Type definition

```
typedef void (USB_MSD_STORAGE_INIT)(      U8                          Lun,
                                    const USB_MSD_INST_DATA_DRIVER * pDriverData);
```

Parameters

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
pDriverData	Pointer to a USB_MSD_INST_DATA_DRIVER structure that contains all information that is necessary for the driver initialization. Refer to USB_MSD_INST_DATA_DRIVER structure for detailed information.

 © 2010-2026 SEGGER Microcontroller GmbH

7.4.3.2 USB_MSD_STORAGE_GETINFO

Description

Retrieves storage medium information such as sector size and number of sectors available.

Type definition

```
typedef void (USB_MSD_STORAGE_GETINFO)(U8             Lun,
                                       USB_MSD_INFO * pInfo);
```

Parameters

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
pInfo	Pointer to a USB_MSD_INFO structure. For detailed information about the USB_MSD_INFO structure, refer to USB_MSD_INFO.

 © 2010-2026 SEGGER Microcontroller GmbH

7.4.3.3 USB_MSD_STORAGE_GETREADBUFFER

Description

Prepares the read function and returns a pointer to a buffer that is used by the storage driver.

Type definition

```
typedef U32 (USB_MSD_STORAGE_GETREADBUFFER)(U8      Lun,
                                             U32     SectorIndex,
                                             void ** ppData,
                                             U32     NumSectors);
```

Parameters

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
SectorIndex	Specifies the start sector for the read operation.
ppData	Pointer to a pointer to store the read buffer address of the driver.
NumSectors	Number of sectors to read.

Return value

Maximum number of consecutive sectors that can be read at once by the driver.

 © 2010-2026 SEGGER Microcontroller GmbH

7.4.3.4 USB_MSD_STORAGE_READ

Description

Reads one or multiple consecutive sectors from the storage medium.

Type definition

```
typedef I8 (USB_MSD_STORAGE_READ)(U8     Lun,
                                  U32    SectorIndex,
                                  void * pData,
                                  U32    NumSectors);
```

Parameters

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
SectorIndex	Specifies the start sector from where the read operation is started.
pData	Pointer to buffer to store the read data.
NumSectors	Number of sectors to read.

Return value

= 0 Success.
≠ 0 Failed.

 © 2010-2026 SEGGER Microcontroller GmbH

7.4.3.5 USB_MSD_STORAGE_GETWRITEBUFFER

Description

Prepares the write function and returns a pointer to a buffer that is used by the storage driver.

Type definition

```
typedef U32 (USB_MSD_STORAGE_GETWRITEBUFFER)(U8      Lun,
                                              U32     SectorIndex,
                                              void ** ppData,
                                              U32     NumSectors);
```

Parameters

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
SectorIndex	Specifies the start sector for the write operation.
ppData	Pointer to a pointer to store the write buffer address of the driver.
NumSectors	Number of sectors to write.

Return value

Maximum number of consecutive sectors that can be written into the buffer.

 © 2010-2026 SEGGER Microcontroller GmbH

7.4.3.6 USB_MSD_STORAGE_WRITE

Description

Writes one or more consecutive sectors to the storage medium.

Type definition

```
typedef I8 (USB_MSD_STORAGE_WRITE)(      U8     Lun,
                                         U32    SectorIndex,
                                   const void * pData,
                                         U32    NumSectors);
```

Parameters

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.
SectorIndex	Specifies the start sector for the write operation.
pData	Pointer to data to be written to the storage medium.
NumSectors	Number of sectors to write.

Return value

= 0 Success.
≠ 0 Failed.

 © 2010-2026 SEGGER Microcontroller GmbH

7.4.3.7 USB_MSD_STORAGE_MEDIUMISPRESENT

Description

Checks if medium is present.

Type definition

```
typedef I8 (USB_MSD_STORAGE_MEDIUMISPRESENT)(U8 Lun);
```

Parameters

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.

Return value

1 Medium is present.
0 Medium is not present.

 © 2010-2026 SEGGER Microcontroller GmbH

7.4.3.8 USB_MSD_STORAGE_DEINIT

Description

De-initializes the storage medium.

Type definition

```
typedef void (USB_MSD_STORAGE_DEINIT)(U8 Lun);
```

Parameters

Parameter	Description
Lun	Logical unit number. Specifies for which drive the function is called.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 8

Virtual Mass Storage Component (VirtualMSD)

This chapter gives a general overview of the VirtualMSD component and describes how to get the VirtualMSD running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

8.1 Overview

The VirtualMSD component allows you to easily stream files to and from USB devices. Once the USB device is connected to the host, files can be read or written to the application without the need for dedicated storage memory.

This makes the software very flexible: it can be used for various types of applications and purposes, with no additional software or drivers necessary on the host side.

The VirtualMSD software analyzes what operation is performed by the host and passes it to the application layer of the embedded target, which then performs the appropriate action. A simple drag-and-drop is all it takes to initialize this process, which is supported by a unique active file technology.

VirtualMSD can access all data that has been created prior to the device being attached to the host; live data cannot be provided.

VirtualMSD allows using the storage device in a virtual manner, which means data does not need to be stored on a physical medium.

The storage device will be shown on the host as a FAT formatted volume with a configurable size and a configurable file list.

With the help of that virtual function, the target device can be used for different applications by simply dragging and dropping files to and from the storage medium:

- Firmware update application.
- Configuration updater.
- File system firewall - protect the target’s file system from being manipulated by the host.

The component itself is based on the MSD class and thus can be used on virtually any OS such as Windows, macOS, or any Linux distribution (including Android) that supports MSD, without installing any third-party tools.

 © 2010-2026 SEGGER Microcontroller GmbH

8.2 Configuration

8.2.1 Initial configuration

To get emUSB-Device-VirtualMSD up and running and to perform an initial test, the configuration as delivered should not be modified.

8.2.2 Final configuration

The configuration must only be modified if emUSB-Device is deployed in your final product. Refer to *emUSB-Device Configuration* on page 53 for detailed information about the generic information functions which must be adapted.

8.2.3 Class specific configuration functions

For basic configuration, please refer to the MSD chapter *MSD class specific configuration functions* on page 252. In addition to the MSD configuration functions described there, the following VirtualMSD functions are available.

Function	Description
emUSB-Device-VirtualMSD configuration functions	
`USB_VMSD_X_Config()`	Configures the VirtualMSD component.

 © 2010-2026 SEGGER Microcontroller GmbH

8.2.3.1 USB_VMSD_X_Config()

Description

Main user configuration function of the VirtualMSD component. This function is provided by the user.

Prototype

```
void USB_VMSD_X_Config(void);
```

Example

```
void USB_VMSD_X_Config(void) {
  //
  // String information used when inquiring the volume.
  //
  static const USB_MSD_LUN_INFO _LunInfo = {
    "Vendor",      // MSD VendorName
    "MSD Volume", // MSD ProductName
    "1.00",        // MSD ProductVer
    "134657890"    // MSD SerialNo
  };
  //
  // Global configuration
  //
  USBD_VMSD_AssignMemory(&_aMEMBuffer[0], sizeof(_aMEMBuffer));
  //
  // Setup LUN0
  //
  USBD_VMSD_SetNumSectors(0, 8000);
  USBD_VMSD_SetSectorsPerCluster(0, 32); // Anywhere from 1...128, needs to be
 2^x
  USBD_VMSD_SetNumRootDirSectors(0, 2);
  USBD_VMSD_SetUserAPI(0, &_UserFuncAPI);
  USBD_VMSD_SetVolumeInfo(0, "Virt0.MSD", &_LunInfo);   // Add volume ID
  //
  // Push const contents to the volume
  //
  USBD_VMSD_AddConstFiles(0, &_aConstFiles[0], COUNTOF(_aConstFiles));
}
```

Additional information

During the call of `USBD_VMSD_Add()`, this user function is called in order to configure the VirtualMSD module according to the user's preferences. In order to allow the user to configure the volume, it is necessary to provide either a memory block or memory allocation/free callbacks to the VirtualMSD component.

8.2.4 Running the example application

The directory `Application` contains example applications that can be used with emUSB-Device and the VirtualMSD component. To test the VirtualMSD component, build and download the application of choice into the target. Remove the USB connection and reconnect the target to the host. The target will enumerate and can be accessed via a file browser.

 © 2010-2026 SEGGER Microcontroller GmbH

8.2.5 Calculation of RAM memory usage for VirtualMSD

An application has to provide RAM memory in order to use VirtualMSD either via a call to the function `USB_VirtualMSD_AssignMemory()` or by setting callback functions for memory allocation. The amount of memory used can be calculated as follows:

For each volume:

Purpose	Bytes used	Minimum
Global volume information	128	128
Cluster info for predefined files added with `USB_VirtualMSD_AddConstFiles()`	2 (for each file)	0
I/O Buffer	512	512
Directory	m * 512	512
FAT	n * 512	512
Total	-	1664

The number of files that can be stored on the volume depends on the size of the directory which is configured using `USB_VirtualMSD_SetNumRootDirSectors()`:

Number of root directory sectors (m)	Used memory for directory (bytes)	max. number of files with short (8.3) file name
1	512	15
2	1024	31
3	1536	47
4	2048	63
5	2560	79
6	3072	95

Files with long file names may occupy multiple entries in the directory, depending on the actual length.

The number of FAT sectors (n) depends on the virtual size of the volume (configured using `USB_VirtualMSD_SetNumSectors()`) and the number of sectors per cluster:

Number of sectors	Sectors per cluster	Used memory for FAT (bytes)	approx. virtual volume size (MB)
10880	32	512	5.4
21792	32	1024	10.8
32704	32	1536	16.3
43616	32	2048	21.7
54528	32	2560	27.2
65440	32	3072	32.7
76352	32	3584	38.1
87264	32	4096	43.6
98176	32	4608	49.0
109088	32	5120	54.5
120000	32	5632	59.9
130720	32	6144	65.3
43520	128	512	21.3
87168	128	1024	43.5

 © 2010-2026 SEGGER Microcontroller GmbH

Number of sectors	Sectors per cluster	Used memory for FAT (bytes)	approx. virtual volume size (MB)
130816	128	1536	65.3
174464	128	2048	87.1
218112	128	2560	108.9
261760	128	3072	130.8
305408	128	3584	152.6
349056	128	4096	174.4
392704	128	4608	196.2
436352	128	5120	218.1
480000	128	5632	239.9
522800	128	6144	261.3

There is no disadvantage to using the maximum possible number of sectors per cluster (128).

In most cases, the minimal configuration (FAT = 512 and directory = 512) should be sufficient. It supports a small number of files with a total size of up to 21 MB. If more files or bigger files are needed, the required parameters can be looked up in the tables above.

 © 2010-2026 SEGGER Microcontroller GmbH

8.3 Target API

Function	Description
API functions	
USBD_VMSD_Add()	Create VirtualMSD volumes and add MSD interface to the device.
User-supplied functions	
USB_VMSD_X_Config()	User supplied function that configures all storages of the VMSD component.
Configuration functions	
USBD_VMSD_AssignMemory()	Assigns memory to the VirtualMSD module.
USBD_VMSD_SetUserAPI()	Sets the default user callbacks for the VirtualMSD component.
USBD_VMSD_SetNumRootDirSectors()	Sets the number of sectors which should be used for root directory entries.
USBD_VMSD_SetVolumeInfo()	Sets the volume name for a specified LUN.
USBD_VMSD_AddConstFiles()	Adds constant files to VirtualMSD.
USBD_VMSD_SetNumSectors()	Sets the number of sectors available on the volume.
USBD_VMSD_SetSectorsPerCluster()	Set number of sectors per cluster.
Data structures	
USB_VMSD_CONST_FILE	This structure contains information about a constant file which cannot be changed at run time and should be shown inside the VirtualMSD volume (e.g.
USB_VMSD_USER_FUNC_API	This structure contains the function pointers for user provided functions.
USB_VMSD_FILE_INFO	Structure used in the read and write callbacks.
USB_VMSD_DIR_ENTRY_SHORT	Structure used to describe an entry with a short file name.
Function definitions	
USB_VMSD_ON_READ_FUNC	Callback function prototype that is used when calling the USBD_VMSD_SetUserAPI() function.
USB_VMSD_ON_WRITE_FUNC	Callback function prototype that is used when calling the USBD_VMSD_SetUserAPI() function.
USB_VMSD_MEM_ALLOC	Function prototype that is used when memory is being allocated by the VirtualMSD module.
USB_VMSD_MEM_FREE	Function prototype that is used when memory is being freed by the VirtualMSD module.

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.1 API functions

8.3.1.1 USBD_VMSD_Add()

Description

Create VirtualMSD volumes and add MSD interface to the device.

Prototype

```
void USBD_VMSD_Add(void);
```

Additional information

After the initialization of emUSB-Device, this is the first function that needs to be called when the VirtualMSD component is used with emUSB-Device. During the call of this function, the user function `USB_VMSD_X_Config()` is called in order to configure the storage itself.

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.1.2 USB_VMSD_X_Config()

Description

User supplied function that configures all storages of the VMSD component.

Prototype

```
void USB_VMSD_X_Config(void);
```

Additional information

This function is called automatically by `USBD_VMSD_Add()` in order to allow to configure the storage volumes that VirtualMSD should show after configuration.

Only the following functions must be called in this context:

Allowed functions with `USB_X_VMSD_Config`:
`USBD_VMSD_AssignMemory()`
`USBD_VMSD_SetUserAPI()`
`USBD_VMSD_SetNumRootDirSectors()`
`USBD_VMSD_SetVolumeInfo()`
`USBD_VMSD_AddConstFiles()`
`USBD_VMSD_SetNumSectors()`
`USBD_VMSD_SetSectorsPerCluster()`

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.1.3 USBD_VMSD_AssignMemory()

Description

Assigns memory to the VirtualMSD module.

Prototype

```
void USBD_VMSD_AssignMemory(U32 * p,
                            U32   NumBytes);
```

Parameters

Parameter	Description
p	Pointer to the memory which should be dedicated to VirtualMSD.
NumBytes	Size of the memory block in bytes.

Additional information

See *Calculation of RAM memory usage for VirtualMSD* on page 294.

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.1.4 USBD_VMSD_SetUserAPI()

Description

Sets the default user callbacks for the VirtualMSD component.

Prototype

```
void USBD_VMSD_SetUserAPI(const USB_VMSD_USER_FUNC_API * pUserFunc);
```

Parameters

Parameter	Description
pUserFunc	Pointer to a USB_VMSD_USER_FUNC_API structure which holds the default function pointers for multiple functions.

Notes

(1) Must only be called from USB_VMSD_X_Config() during initialization phase

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.1.5 USBD_VMSD_SetNumRootDirSectors()

Description

Sets the number of sectors which should be used for root directory entries.

Prototype

```
void USBD_VMSD_SetNumRootDirSectors(unsigned Lun,
                                    unsigned NumRootDirSectors);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
NumRootDirSectors	Number of sectors to be reserved for the root directory entries.

Additional information

The number of sectors reserved through this function is subtracted from the number of sectors configured by `USBD_VMSD_SetNumSectors()`. These sectors hold the root directory entries for the specified LUN. A single sector contains 512 bytes, a short file name entry (also called 8.3 filenames) needs 32 bytes, therefore a single sector has enough space for 16 root directory entries. Please note that when using LFN (long file names) the number of entries required for a single file is dynamic (depending on the length of the file name).

Notes

(1) Must only be called from `USB_VMSD_X_Config()` during initialization phase

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.1.6 USBD_VMSD_SetVolumeInfo()

Description

Sets the volume name for a specified LUN.

Prototype

```
int USBD_VMSD_SetVolumeInfo(      unsigned           Lun,
                            const char             * sVolumeName,
                            const USB_MSD_LUN_INFO * pLunInfo);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
sVolumeName	Pointer to a string containing the name of the LUN.
pLunInfo	Pointer to USB_MSD_LUN_INFO structure containing all relevant MSD strings.

Return value

≥ 0 O.K.
< 0 Error

Notes

(1) Must only be called from USB_VMSD_X_Config() during initialization phase

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.1.7 USBD_VMSD_AddConstFiles()

Description

Adds constant files to VirtualMSD. Allows to add multiple files which should be shown on a VirtualMSD volume as soon as it is connected. A common example would be a "Readme.txt" or a link to the company website.

Prototype

```
int USBD_VMSD_AddConstFiles(      unsigned                Lun,
                            const USB_VMSD_CONST_FILE * paConstFile,
                                  unsigned                NumFiles);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
paConstFile	Pointer to an array of USB_VMSD_CONST_FILE structures.
NumFiles	The number of items in the paConstFile array.

Return value

≥ 0 O.K.
< 0 Error

Additional information

For additional information please see USB_VMSD_CONST_FILE.

Notes

(1) Must only be called from USB_VMSD_X_Config() during initialization phase

Example

```
#define COUNTOF(a)          (sizeof((a))/sizeof((a)[0]))
static  const  U8  _abFile_SeggerHTML[]  =  {0x3C,  0x68,  0x74,  0x6D,  0x6C,  0x3E,  0x3C,
0x68, 0x65, 0x61, 0x64, 0x3E, 0x3C, 0x6D, 0x65, 0x74, 0x61, 0x20, 0x68, 0x74, 0x74,
0x70, 0x2D, 0x65, 0x71, 0x75, 0x69, 0x76, 0x3D, 0x22, 0x72, 0x65, 0x66, 0x72, 0x65,
0x73, 0x68, 0x22, 0x20, 0x63, 0x6F, 0x6E, 0x74, 0x65, 0x6E, 0x74, 0x3D, 0x22, 0x30,
0x3B, 0x20, 0x75, 0x72, 0x6C, 0x3D, 0x68, 0x74, 0x74, 0x70, 0x3A, 0x2F, 0x2F, 0x77,
0x77, 0x77, 0x2E, 0x73, 0x65, 0x67, 0x67, 0x65, 0x72, 0x2E, 0x63, 0x6F, 0x6D, 0x2F,
0x69, 0x6E, 0x64, 0x65, 0x78, 0x2E, 0x68, 0x74, 0x6D, 0x6C, 0x22, 0x2F, 0x3E, 0x3C,
0x74, 0x69, 0x74, 0x6C, 0x65, 0x3E, 0x53, 0x45, 0x47, 0x47, 0x45, 0x52, 0x20, 0x53,
0x68, 0x6F, 0x72, 0x74, 0x63, 0x75, 0x74, 0x3C, 0x2F, 0x74, 0x69, 0x74, 0x6C, 0x65,
0x3E, 0x3C, 0x2F, 0x68, 0x65, 0x61, 0x64, 0x3E, 0x3C, 0x62, 0x6F, 0x64, 0x79, 0x3E,
0x3C, 0x2F, 0x62, 0x6F, 0x64, 0x79, 0x3E, 0x3C, 0x2F, 0x68, 0x74, 0x6D, 0x6C, 0x3E};
static USB_VMSD_CONST_FILE _aConstFiles[] = {
// sName             pData                FileSize                      Flags
  { "Segger.html",   _abFile_SeggerHTML,    sizeof(_abFile_SeggerHTML),    0, }
};
/*********************************************************************
*
*       USB_VMSD_X_Config
*
*  Function description
*    This function is called by the USB MSD Module during USB_VMSD_Init() and
initializes the VirtualMSD volume.
*/
void USB_VMSD_X_Config(void) {
  <...>
  USBD_VMSD_AddConstFiles(1, &_aConstFiles[0], COUNTOF(_aConstFiles));
  <...>
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.1.8 USBD_VMSD_SetNumSectors()

Description

Sets the number of sectors available on the volume.

Prototype

```
void USBD_VMSD_SetNumSectors(unsigned Lun,
                             unsigned NumSectors);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
NumSectors	Specifies the number of sectors for a LUN.

Notes

(1) Must only be called from USB_VMSD_X_Config() during initialization phase

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.1.9 USBD_VMSD_SetSectorsPerCluster()

Description

Set number of sectors per cluster.

Prototype

```
void USBD_VMSD_SetSectorsPerCluster(unsigned Lun,
                                    unsigned SectorsPerCluster);
```

Parameters

Parameter	Description
Lun	Specifies the logical unit number.
SectorsPerCluster	Number of sectors per cluster for the LUN.

Additional information

SectorsPerCluster can be anywhere between 1 and 128, but needs to be a power of 2. Larger clusters save memory because the management overhead is lower, but the maximum number of files is limited by the number of available clusters.

Notes

(1) Must only be called from USB_VMSD_X_Config() during initialization phase

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.2 Data structures

8.3.2.1 USB_VMSD_CONST_FILE

Description

This structure contains information about a constant file which cannot be changed at run time and should be shown inside the VirtualMSD volume (e.g. Readme.txt). This structure is a parameter for the `USBD_VMSD_AddConstFiles()` function.

Type definition

```
typedef struct {
  const char  * sName;
  const U8    * pData;
  unsigned      FileSize;
  U32           Flags;
} USB_VMSD_CONST_FILE;
```

Structure members

Member	Description
`sName`	Pointer to a zero-terminated string containing the filename.
`pData`	Pointer to the file data. Can be `NULL`.
`FileSize`	Size of the file. Normally the size of the data pointed to by `pData`.
`Flags`	Can be one of the following items: • `USB_VMSD_FILE_WRITABLE`: The file is writable • `USB_VMSD_FILE_AHEAD`: File is located at the start of the volume. Normally constant files are allocated at the end of the volume.

Additional information

If a file does not occupy complete sectors the remaining bytes of the last sector are automatically filled with 0s on read. If `pData` is `NULL` the file is not displayed in the volume. This is useful when the application has certain files which should only be displayed after certain events (e.g. the application displays a Fail.txt when the device is reconnected after an unsuccessful firmware update).

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.2.2 USB_VMSD_USER_FUNC_API

Description

This structure contains the function pointers for user provided functions. This structure is a parameter for the USBD_VMSD_SetUserAPI() function.

Type definition

```
typedef struct _USB_VMSD_USER_FUNC_API {
  USB_VMSD_ON_READ_FUNC    * pfOnReadSector;
  USB_VMSD_ON_WRITE_FUNC   * pfOnWriteSector;
  USB_VMSD_MEM_ALLOC       * pfMemAlloc;
  USB_VMSD_MEM_FREE        * pfMemFree;
} USB_VMSD_USER_FUNC_API;
```

Structure members

Member	Description
pfOnReadSector	Pointer to a callback function of type USB_VMSD_ON_READ_FUNC which is called when a sector is read from the host. This function is mandatory and can not be NULL.
pfOnWriteSector	Pointer to a callback function of type USB_VMSD_ON_WRITE_FUNC which is called when a sector is written from the host. This function is mandatory and can not be NULL.
pfMemAlloc	Pointer to a user provided alloc function of type USB_VMSD_MEM_ALLOC. If this pointer is NULL the internal alloc function is called. If no memory block is assigned USB_PANIC() is called.
pfMemFree	Pointer to a user provided free function of type USB_VMSD_MEM_FREE. If this pointer is NULL the internal free function is called.

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.2.3 USB_VMSD_FILE_INFO

Description

Structure used in the read and write callbacks.

Type definition

```
typedef struct {
  const USB_VMSD_DIR_ENTRY_SHORT  * pDirEntry;
} USB_VMSD_FILE_INFO;
```

Structure members

Member	Description
pDirEntry	Pointer to a USB_VMSD_DIR_ENTRY_SHORT structure.

Additional information

Check USB_VMSD_ON_READ_FUNC, USB_VMSD_ON_WRITE_FUNC and USB_VMSD_DIR_ENTRY_SHORT for more information.

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.2.4 USB_VMSD_DIR_ENTRY_SHORT

Description

Structure used to describe an entry with a short file name. This structure is a member of USB_VMSD_DIR_ENTRY.

Type definition

```
typedef struct {
  U8    acFilename[8];
  U8    acExt[3];
  U8    DirAttr;
  U8    NTRes;
  U8    CrtTimeTenth;
  U16   CrtTime;
  U16   CrtDate;
  U16   LstAccDate;
  U16   FstClusHI;
  U16   WrtTime;
  U16   WrtDate;
  U16   FstClusLO;
  U32   FileSize;
} USB_VMSD_DIR_ENTRY_SHORT;
```

Structure members

Member	Description
acFilename	File name, limited to 8 characters (short file name), padded with spaces (0x20).
acExt	File extension, limited to 3 characters (short file name), padded with spaces (0x20).
DirAttr	File attributes. Available attributes are listed below.
NTRes	Reserved for use by Windows NT.
CrtTimeTenth	Millisecond stamp at file creation time. This field actually contains a count of tenths of a second.
CrtTime	Creation time.
CrtDate	Date file was created.
LstAccDate	Last access date. Note that there is no last access time, only a date. This is the date of last read or write.
FstClusHI	High word of this entry’s first cluster number.
WrtTime	Time of last write.
WrtDate	Date of last write.
FstClusLO	Low word of this entry’s first cluster number.
FileSize	File size in bytes.

Additional information

The following file attributes are available for short dir entries:

Attribute	Explanation
USB_VMSD_ATTR_READ_ONLY	The file is read-only.
USB_VMSD_ATTR_HIDDEN	The file is hidden.
USB_VMSD_ATTR_SYSTEM	The file is designated as a system file.
USB_VMSD_ATTR_VOLUME_ID	This entry is the volume ID (volume name).
USB_VMSD_ATTR_DIRECTORY	The file is a directory.

 © 2010-2026 SEGGER Microcontroller GmbH

Attribute	Explanation
USB_VMSD_ATTR_ARCHIVE	The file has the archive attribute.
USB_VMSD_ATTR_LONG_NAME	The file has a long file name.

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.3 Function definitions

8.3.3.1 USB_VMSD_ON_READ_FUNC

Description

Callback function prototype that is used when calling the USBD_VMSD_SetUserAPI() function.

Type definition

```
typedef int (USB_VMSD_ON_READ_FUNC)(      unsigned                Lun,
                                          U8                    * pData,
                                          U32                     Off,
                                          U32                     NumBytes,
                                    const USB_VMSD_FILE_INFO    * pFile);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one virtual volume, this parameter is 0.
pData	Pointer to a buffer in which the data is stored.
Off	Offset in the file which is read by the host.
NumBytes	Amount of bytes requested by the host.
pFile	Pointer to a USB_VMSD_FILE_INFO structure describing the file.

Return value

= 0 Success.
≠ 0 An error occurred.

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.3.2 USB_VMSD_ON_WRITE_FUNC

Description

Callback function prototype that is used when calling the USBD_VMSD_SetUserAPI() function.

Type definition

```
typedef int (USB_VMSD_ON_WRITE_FUNC)(      unsigned                Lun,
                                     const U8                    * pData,
                                           U32                     Off,
                                           U32                     NumBytes,
                                     const USB_VMSD_FILE_INFO    * pFile);
```

Parameters

Parameter	Description
Lun	Zero-based index for the unit number. Using only one virtual volume, this parameter is 0.
pData	Pointer to the data to be written (received from the host). If pData = NULL, then there are no data written by the host, but instead a new or changed directory entry was written, which is provided via pFile.
Off	Offset in the file which the host writes.
NumBytes	Amount of bytes to write.
pFile	Pointer to a USB_VMSD_FILE_INFO structure describing the file or NULL.

Return value

0	Success.
1	Enable continuous sector mode: From now on, only forward writes to continuous sectors to the user callback. Ignore writes to all other sectors.
−1	Disable continuous sector mode.
−2	Report write error to USB host.

Additional information

Depending on the behavior of the host operating system it is possible that pFile is NULL. In this case we recommend to perform data analysis to recognize the file.

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.3.3 USB_VMSD_MEM_ALLOC

Description

Function prototype that is used when memory is being allocated by the VirtualMSD module.

Type definition

```
typedef void * (USB_VMSD_MEM_ALLOC)(U32 Size);
```

Parameters

Parameter	Description
Size	Size of the required memory in bytes.

Return value

Pointer to the allocated memory or NULL.

 © 2010-2026 SEGGER Microcontroller GmbH

8.3.3.4 USB_VMSD_MEM_FREE

Description

Function prototype that is used when memory is being freed by the VirtualMSD module.

Type definition

```
typedef void (USB_VMSD_MEM_FREE)(void * p);
```

Parameters

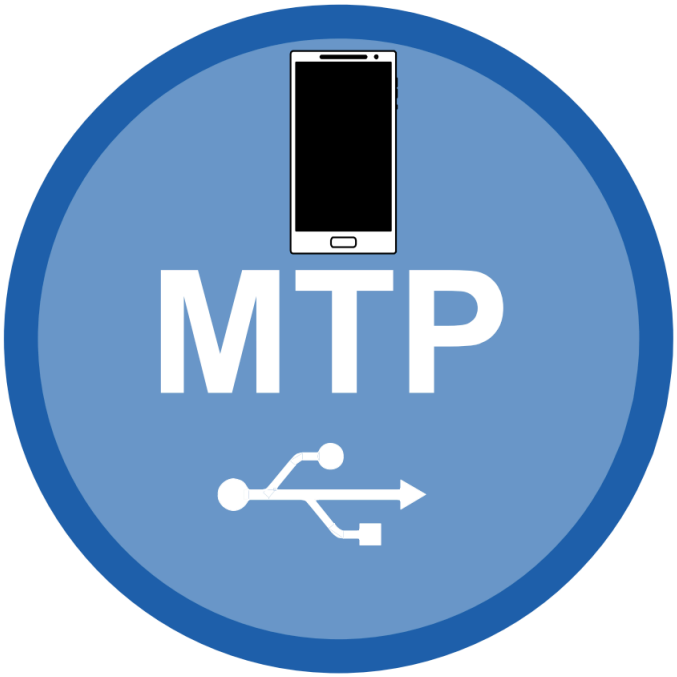
Parameter	Description
p	Pointer to a memory block which was previously allocated by USB_VMSD_MEM_ALLOC.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 9

Media Transfer Protocol Class (MTP)

This chapter gives a general overview of the MTP class and describes how to get the MTP component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

9.1 Overview

The Media Transfer Protocol (MTP) is a USB class protocol which can be used to transfer files to and from storage devices. MTP is an official extension of the Picture Transfer Protocol (PTP) designed to allow digital cameras to exchange image files with a computer. MTP extends this by adding support for arbitrary data types.

MTP is an alternative to Mass Storage Device (MSD) and in contrast to MSD which reads and writes sector data, it operates at the file level. This type of operation gives MTP some advantages over MSD:

- The cable can be safely removed during a data transfer without damaging the file system.
- The file system does not need to be FAT (can be the SEGGER emFile File System (EFS) or any other proprietary file system).
- The application has full control over which files are visible to the user. Selected files or directories can be hidden.
- Virtual files can be presented.
- Host and target can access storage simultaneously without conflicts.

MTP is supported by most operating systems out of the box and the installation of additional drivers is not required.

emUSB-Device-MTP supports the following capabilities:

- File read
- File write
- Format
- File delete
- Directory create
- Directory delete

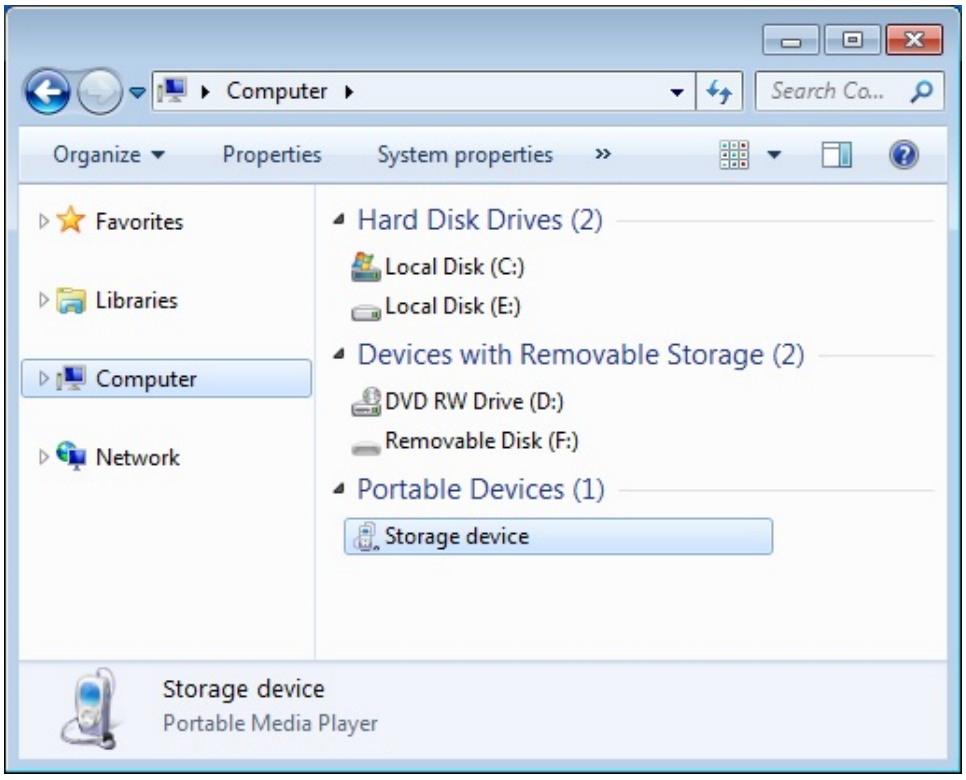
emUSB-Device-MTP comes as a complete package and contains the following:

- Generic USB handling
- MTP device class implementation
- Storage driver which uses emFile
- Sample application showing how to work with MTP

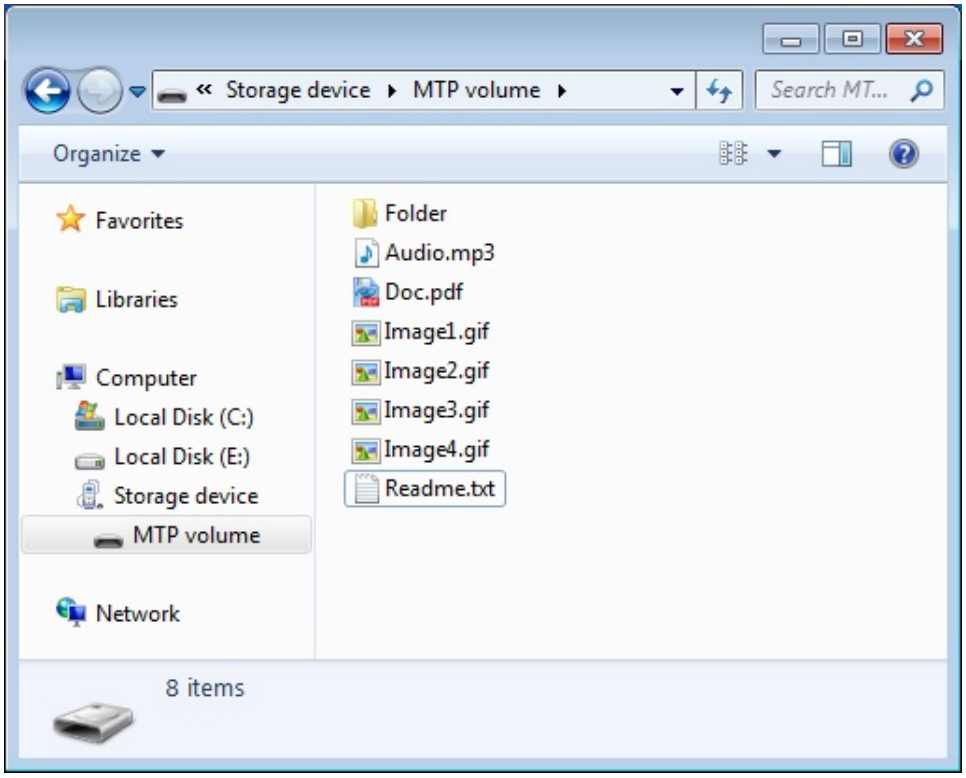
 © 2010-2026 SEGGER Microcontroller GmbH

9.1.1 Getting access to files

An MTP device will be displayed under the "Portable Devices" section in the "Computer" window when connected to a PC running the Microsoft Windows 7 operating system:

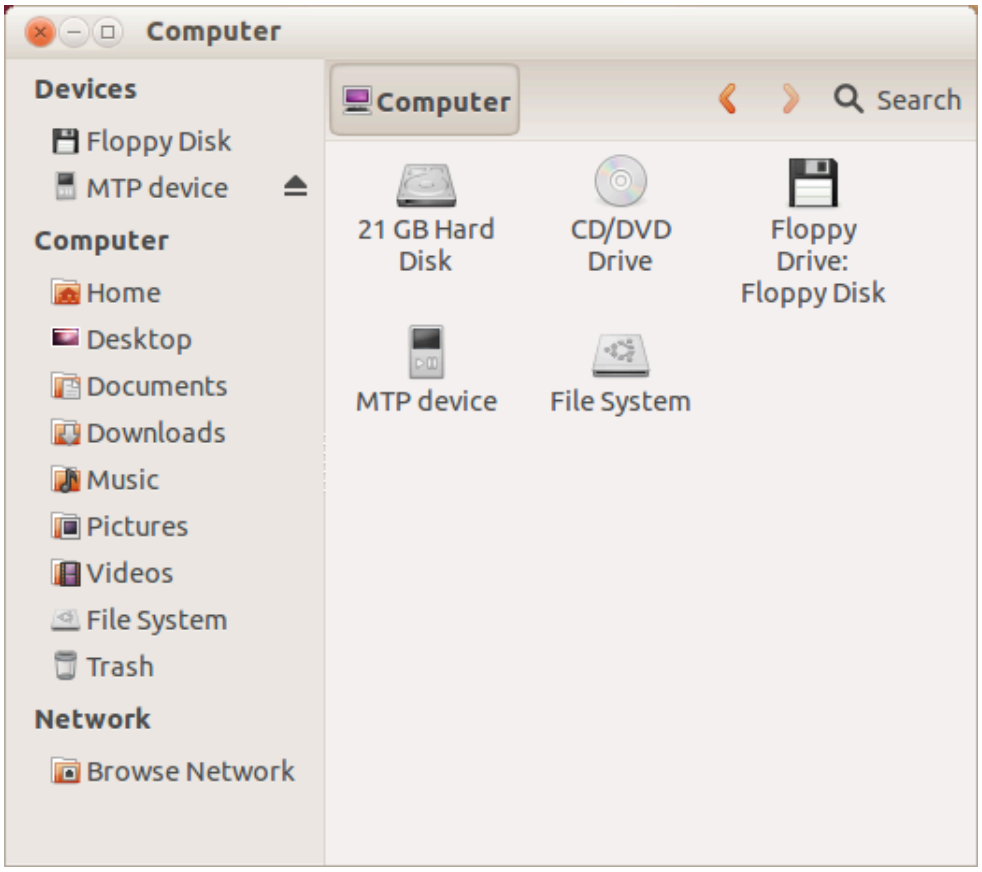


The files and directories stored on the device are accessed in the usual way using the Windows Explorer:



 © 2010-2026 SEGGER Microcontroller GmbH

On the Ubuntu Linux operating system, a connected MTP device is shown in the "Computer" window:



The files and directories present on the MTP device can be easily accessed via the GUI:



On other operating systems, the data stored on MTP devices can be accessed similarly.

 © 2010-2026 SEGGER Microcontroller GmbH

macOS does not support MTP natively; therefore, a third-party tool is necessary. Please take a closer look at our wiki pages on this topic.

 © 2010-2026 SEGGER Microcontroller GmbH

9.1.2 Additional information

For more technical details about MTP and PTP follow these links:

MTP specification

PTP specification

9.1.3 Error handling on the host side

It seems that the MTP module on Windows has been changed from Windows 7/8.1 to Windows 10/11, with the result that the error handling no longer works properly, at least when MTP is used with Windows Explorer. In some cases, no error message is shown; the operation is just aborted. Self-written tools or any third-party tool, such as Multi-Commander Multi-Commander, that natively support MTP (referred to as portable devices) show proper error messages. On Linux, there are different ways to integrate MTP with the root file system, either by using a FUSE implementation (such as mtpfs or jmtpfs) or by using extensions for the desktop manager/file manager, such as gvfs. In most cases, the add-on is based on libmtp. Depending on the version, you may get different results when an error is passed from the MTP device to the host (libmtp).

 © 2010-2026 SEGGER Microcontroller GmbH

9.2 Configuration

9.2.1 Initial configuration

To get emUSB-Device-MTP up and running as well as to do an initial test, the configuration as delivered with the sample application should not be modified.

9.2.2 Final configuration

The configuration must only be modified when emUSB-Device is integrated in your final product. Refer to section *emUSB-Device Configuration* on page 53 for detailed information about the generic information functions which have to be adapted.

9.2.3 emFile and MTP configuration for UTF8 characters

If you need to support non-ASCII characters, you need to set the `MTP_SUPPORT_UTF8` define to 1 in your `USB_Conf.h` file. Furthermore, you need to set the `FS_SUPPORT_FILE_NAME_ENCODING`, `FS_SUPPORT_EXT_ASCII`, and `FS_SUPPORT_MBCS` defines to 1 in your `FS_Conf.h` file. Additionally, you must make sure that LFN is active (`FS_FAT_SupportLFN()`) and that the following functions have been called: `FS_SetCharSetType(&FS_CHARSET_CP932)`, `FS_FAT_SetLFNConverter(&FS_UNICODE_CONV_UTF8)`. See the emFile documentation for details.

If you are not using emFile, you must make sure that your file system is using UTF8.

9.2.4 Class specific configuration

Besides the generic emUSB-Device configuration functions (*emUSB-Device Configuration* on page 53), the following should be adapted before the emUSB-Device MTP component is used in a final product. Example implementations are supplied in the MSD example application `USB_MTP_Start.c`, located in the `Application` directory of emUSB-Device.

An MTP device is required to present an additional information set to the host. These values are added during the initial call to `USBD_MTP_Add()`.

Example

```
static const USB_MTP_INFO _MTPInfo = {
  "Vendor",          // MTP Manufacturer
  "Storage device", // MTP Model
  "1.0",             // MTP DeviceVersion
  "0123456789ABCDEF0123456789ABCDEF" // MTP SerialNumber.
                                     // It must be exactly 32 characters long.
};
...
InitData.pMTPInfo = &_MTPInfo;
...
USB_MTP_Add(&InitData);
```

9.2.5 Compile time configuration

The following macros can be added to the `USB_Conf.h` file in order to configure the behavior of the MTP component.

The following types of configuration macros exist:

Binary switches "B"

Switches can have a value of either 0 or 1, for deactivated and activated respectively. Actually, anything other than 0 works, but 1 makes it easier to read a configuration file. These switches can enable or disable a certain functionality or behavior. Switches are the simplest form of configuration macros.

 © 2010-2026 SEGGER Microcontroller GmbH

Numerical values "N"

Numerical values are used somewhere in the code in place of a numerical constant.

Type	Macro	Default	Description
N	MTP_MAX_NUM_STORAGES	4	Maximum number of storage units the storage layer can handle. 4 additional bytes are allocated for each storage unit.
B	MTP_SAVE_FILE_INFO	0	Specifies if the object properties (file size, write protection, creation date, modification date and file ID) should be stored in RAM for quick access to them. This can have a noticeable impact on displaying folders with a large amount of objects in them. With this switch set to 0, objects require 12 bytes + the size of the file name inside the object list memory area. 33 additional bytes of RAM are required for each object when the switch is set to 1.
N	MTP_MAX_FILE_PATH	256	Maximum number of characters in the path to a file or directory.
B	MTP_SUPPORT_UTF8	1	Names of the files and directories which are exchanged between the MTP component and the file system are encoded in UTF-8 format.
B	MTP_SUPPORT_EVENTS	1	Support Events such as object removed/added, new storage added/removed.
B	USB_MTP_NAME_CASE_SENSITIVE	0	When checking file names and directory names, the string compare will be case sensitive.
B	USB_MTP_OLD_MOUNTING_BEHAVIOR	0	With version V3.54.0 and older, the MTP Storage layer for emFile would automatically call FS_Mount. This is no longer the case. This define allows you to restore this old behavior. When set to 1, the storage layer will call FS_Mount automatically.

 © 2010-2026 SEGGER Microcontroller GmbH

9.3 Running the sample application

The directory `Application` contains a sample application `USB_MTP_Start.c` which can be used with emUSB-Device and the MTP component. To test the emUSB-Device-MTP component, the application should be built and then downloaded to the target. Remove the USB connection and reconnect the target to the host. The target will enumerate and will be accessible via a file browser.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4 Target API

Function	Description
API functions	
USBD_MTP_Add()	Adds an MTP interface to the USB stack.
USBD_MTP_AddStorage()	Adds a storage device to emUSB-Device-MTP.
USBD_MTP_RemoveStorage()	
USBD_MTP_Task()	Main task function of MTP component which processes the commands from host.
USBD_MTP_Poll()	Function which handles MTP commands.
USBD_MTP_SendEvent()	Sends an event notification to the host.
USBD_MTP_SetObjectAllocFailCb()	Allows to set a callback which is called when the object list is full and new objects can no longer be allocated.
USBD_MTP_SetOperationCb()	Allows to set a callback which is called when operations are executed by the host operating system via MTP.
USBD_MTP_SetWriteProtectionCb()	Allows to set a callback which is called on all write operations (write, rename, create, delete, modify) and allows the application to prevent the operation for selected files.
Data structures	
USB_MTP_FILE_INFO	Structure which stores information about a file or directory.
USB_MTP_INIT_DATA	Structure which stores the parameters of the MTP interface.
USB_MTP_INFO	Structure that is used when initialising the MTP module.
USB_MTP_INST_DATA	Structure which stores the parameters of storage driver.
USB_MTP_INST_DATA_DRIVER	Structure which stores the parameters passed to the storage driver.
USB_MTP_STORAGE_API	Structure that contains callbacks to the storage driver.
USB_MTP_STORAGE_INFO	Structure which stores information about a storage.
USB_MTP_OPERATION_INFO	Structure which provides information about a new MTP operation.
Enums	
USB_MTP_EVENT	Enum containing the MTP event codes.
USB_MTP_OPERATION_CB_TYPE	Enum containing the callback operation types.
USB_MTP_STORAGE_CB_TYPE	Enum containing the callback operation types for the USB_MTP_WP_CB callback registered via USBD_MTP_SetWriteProtectionCb().
Prototypes	
USB_MTP_OBJECT_ALLOC_FAIL	Callback which can be set via USBD_MTP_SetObjectAllocFailCb().

 © 2010-2026 SEGGER Microcontroller GmbH

Function	Description
USB_MTP_OPERATION_CB	Callback which can be set via USBD_MTP_SetOperationCb().
USB_MTP_WP_CB	Callback which can be set via USBD_MTP_SetWriteProtectionCb().

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.1 API functions

9.4.1.1 USBD_MTP_Add()

Description

Adds an MTP interface to the USB stack.

Prototype

```
int USBD_MTP_Add(const USB_MTP_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a USB_MTP_INIT_DATA structure.

Return value

0 - Successfully added.

Additional information

After the initialization of USB core, this is the first function that needs to be called when an MTP interface is used with emUSB-Device. The structure USB_MTP_INIT_DATA has to be initialized before USB_MTP_Add() is called. Refer to USB_MTP_INIT_DATA for more information.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.1.2 USBD_MTP_AddStorage()

Description

Adds a storage device to emUSB-Device-MTP.

Prototype

```
USB_MTP_STORAGE_HANDLE USBD_MTP_AddStorage(const USB_MTP_INST_DATA * pInstData);
```

Parameters

Parameter	Description
pInstData	Pointer to a USB_MTP_INST_DATA structure which contains the parameters of the added storage.

Return value

= 0 Invalid handle, storage could not be added
≠ 0 A valid storage handle, this handle can be used with the USBD_MTP_SendEvent to indicate an event to the host.

Additional information

It is necessary to call this function immediately after USBD_MTP_Add() and before USBD_MTP_Task()/USBD_MTP_Poll() is called. This function adds a storage device such as a hard drive, MMC/SD card or NAND flash etc., to emUSB-Device-MTP, which will be used as source/destination of data exchange with the host. The structure USB_MTP_INST_DATA must be initialized before USB_MTP_AddStorage() is called. Refer to USB_MTP_INST_DATA for more information.

If a storage was removed in the middle of operation via USBD_MTP_RemoveStorage() it can be added again by calling this function with the same parameters. Additionally the host must be informed of the change by calling USBD_MTP_SendEvent(Handle, USB_MTP_EVENT_STOREADDED, NULL)

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.1.3 USBD_MTP_RemoveStorage()

Prototype

```
int USBD_MTP_RemoveStorage(USB_MTP_STORAGE_HANDLE hStorage);
```

Parameters

Parameter	Description
hStorage	Valid storage handle.

Return value

= 0 Storage removed
≠ 0 An error occurred.

Additional information

It is necessary to notify the host about the storage removal through an MTP event prior to calling this function. The following call can be used: `USBD_MTP_SendEvent(Handle, USB_MTP_EVENT_STOREREMOVED, NULL)`

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.1.4 USBD_MTP_Task()

Description

Main task function of MTP component which processes the commands from host.

Prototype

```
void USBD_MTP_Task(void);
```

Additional information

The `USBD_MTP_Task()` should be called after the USB device has been successfully enumerated and configured. The function returns when the USB device is detached or suspended.

Check `USBD_MTP_Poll()` if you need a non-blocking version.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.1.5 USBD_MTP_Poll()

Description

Function which handles MTP commands. Using this function is only necessary if you want to avoid using the blocking `USB_MTP_Task` function. This can be necessary if you are not using an RTOS.

Prototype

```
int USBD_MTP_Poll(void);
```

Return value

2	O.K. Command was processed, but a protocol error occurred.
1	O.K. Command was processed successfully.
0	O.K. Timeout occurred.
	1 An error occurred. (e.g. no cable connected).

Additional information

This function must be called periodically, otherwise the host can time out the device since it does not process commands. It normally blocks for `USB_MTP_POLL_TIMEOUT` milliseconds while waiting for a command from the host. Should a command arrive during the timeout it will be processed (and the function will block for the complete duration of the command), which could potentially increase the block duration. The duration could also decrease because the function returns as soon as a command is finished.

In case of return value 2 the calling task should still call this function again, if possible recovery will be initiated.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.1.6 USBD_MTP_SendEvent()

Description

Sends an event notification to the host.

Prototype

```
int USBD_MTP_SendEvent(USB_MTP_STORAGE_HANDLE   hStorage,
                       USB_MTP_EVENT            Event,
                       void                   * pPara);
```

Parameters

Parameter	Description
hStorage	Handle to a storage that was returned by USBD_MTP_AddStorage().
Event	Event that occurred. The following events are currently supported: • USB_MTP_EVENT_OBJECTADDED • USB_MTP_EVENT_OBJECTREMOVED • USB_MTP_EVENT_STOREADDED • USB_MTP_EVENT_STOREREMOVED • USB_MTP_EVENT_OBJECTINFOCHANGED • USB_MTP_EVENT_STOREFULL • USB_MTP_EVENT_STORAGEINFOCHANGED
pPara	Pointer to additional information. This parameter depends on the event. In case of Event = • USB_MTP_EVENT_OBJECTADDED • USB_MTP_EVENT_OBJECTREMOVED • USB_MTP_EVENT_OBJECTINFOCHANGED pPara is a pointer to a filled USB_MTP_FILE_INFO structure. • USB_MTP_EVENT_STOREADDED • USB_MTP_EVENT_STOREREMOVED • USB_MTP_EVENT_STORAGEINFOCHANGED pPara is not used and can be NULL.

Return value

= 0 Event sent out successfully.
≠ 0 Event could not be sent.

Additional information

Sending an event notification to the MTP host makes sure that the MTP host is aware of changes in the file system of the storage. This function can also be used to notify that a storage has been added or removed. The events USB_MTP_EVENT_STOREREMOVED and USB_MTP_EVENT_STOREADDED do not affect the internal object list.

Example

```
static void _GetFileInfo(const char * sPath, USB_MTP_FILE_INFO * pFileInfo) {
  const char * s;
  U8 AttrFS;
  U8 AttrMTP;

  memset(pFileInfo, 0, sizeof(USB_MTP_FILE_INFO));
  s = strrchr(sPath, '\\');
  if (s) {
    s++; // Go to the next character after '\'.
  } else {
    s = sPath;
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
  }
  //
  // In case the file path starts with \ skip this.
  //
  if (*sPath == '\\') {
    sPath++;
  }
  pFileInfo->pFileName = (char *)s;
  pFileInfo->pFilePath = (char *)sPath;
  FS_GetFileTimeEx(pFileInfo->pFilePath, &pFileInfo->CreationTime,
                   FS_FILETIME_CREATE);
  FS_GetFileTimeEx(pFileInfo->pFilePath, &pFileInfo->LastWriteTime,
                   FS_FILETIME_MODIFY);
  pFileInfo->IsDirectory = 0;
  AttrFS = FS_GetFileAttributes(pFileInfo ? pFilePath);
  if (AttrFS & FS_ATTR_DIRECTORY) {
    pFileInfo->IsDirectory = 1;
  }
  AttrMTP = 0;
  if (AttrFS & FS_ATTR_READ_ONLY) {
    AttrMTP |= MTP_FILE_ATTR_WP;
  }
  if (AttrFS & FS_ATTR_SYSTEM) {
    AttrMTP |= MTP_FILE_ATTR_SYSTEM;
  }
  if (AttrFS & FS_ATTR_HIDDEN) {
    AttrMTP |= MTP_FILE_ATTR_HIDDEN;
  }
  pFileInfo->Attributes = AttrMTP;
}

static int _WriteLogFile(const char * sLogFilePath) {
  char ac[30];
  FS_FILE * pFile;
  int r = 0;
  USB_MTP_FILE_INFO FileInfo = {0};

  if (FS_IsVolumeMounted("")) {
    //
    // Check whether file already exists
    //
    pFile = FS_FOpen(sLogFilePath, "r");
    if (pFile) {
      r = USB_MTP_EVENT_OBJECTINFOCHANGED;
      FS_Fclose(pFile);
    } else {
      r = USB_MTP_EVENT_OBJECTADDED;
    }
    pFile = FS_FOpen(sLogFilePath, "a+");
    if (pFile) {
      sprintf(ac, "OS_Time = %.8d\r\n", (int)OS_GetTime());
      FS_Write(pFile, ac, 20);
      FS_Fclose(pFile);
    } else {
      r = 0;
    }
  }
  _GetFileInfo(sLogFilePath, &FileInfo);
  //
  // Send events to the host.
  //
  USBD_MTP_SendEvent(_ahStorage[0], (USB_MTP_EVENT)r, &FileInfo);
  USBD_MTP_SendEvent(_ahStorage[0], USB_MTP_EVENT_STORAGEINFOCHANGED, NULL);
  return r;
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.1.7 USBD_MTP_SetObjectAllocFailCb()

Description

Allows to set a callback which is called when the object list is full and new objects can no longer be allocated. See `USB_MTP_OBJECT_ALLOC_FAIL` for details.

Prototype

```
void USBD_MTP_SetObjectAllocFailCb(USB_MTP_OBJECT_ALLOC_FAIL * pf);
```

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.1.8 USBD_MTP_SetOperationCb()

Description

Allows to set a callback which is called when operations are executed by the host operating system via MTP. See `USB_MTP_OPERATION_CB` for details.

Prototype

```
void USBD_MTP_SetOperationCb(USB_MTP_OPERATION_CB * pf);
```

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.1.9 USBD_MTP_SetWriteProtectionCb()

Description

Allows to set a callback which is called on all write operations (write, rename, create, delete, modify) and allows the application to prevent the operation for selected files. See USB_MTP_WP_CB for details.

Prototype

```
void USBD_MTP_SetWriteProtectionCb(USB_MTP_WP_CB * pf);
```

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.2 Data structures

9.4.2.1 USB_MTP_FILE_INFO

Description

Structure which stores information about a file or directory.

Type definition

```
typedef struct USB_MTP_FILE_INFO {
  const char  * pFilePath;
  const char  * pFileName;
  U64           FileSize;
  U32           CreationTime;
  U32           LastWriteTime;
  U8            IsDirectory;
  U8            Attributes;
  U8            acId[MTP_NUM_BYTES_FILE_ID];
} USB_MTP_FILE_INFO;
```

Structure members

Member	Description
pFilePath	Full path to file.
pFileName	Pointer to beginning of file/directory name in pFilePath.
FileSize	Size of the file in bytes.
CreationTime	The time and date when the file was created.
LastWriteTime	The time and date when the file was last modified.
IsDirectory	Set to 1 if the path points to a directory.
Attributes	Bitmask of file attributes (MTP_FILE_ATTR_...).
acId	Unique identifier which persists between MTP sessions.

Additional information

The date and time is formatted as follows:

Bit range	Value range	Description
0-4	0-29	2-second count
5-10	0-59	Minutes
11-15	0-23	Hours
16-20	1-31	Day of month
21-24	1-12	Month of year
25-31	0-127	Number of years since 1980

The following attributes are supported:

Bitmask	Description
MTP_FILE_ATTR_WP	File/directory cannot be modified.
MTP_FILE_ATTR_SYSTEM	File/directory is required for the correct functioning of the system.
MTP_FILE_ATTR_HIDDEN	File/directory should not be shown to the user.

acId should be unique for each file and directory on the file system, and it should be persistent between MTP sessions.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.2.2 USB_MTP_INIT_DATA

Description

Structure which stores the parameters of the MTP interface.

Type definition

```
typedef struct {
  U8                    EPIn;
  U8                    EPOut;
  U8                    EPInt;
  void                * pObjectList;
  U32                   NumBytesObjectList;
  void                * pDataBuffer;
  U32                   NumBytesDataBuffer;
  const USB_MTP_INFO  * pMTPInfo;
  U8                    InterfaceNum;
  U32                   NumBytesAllocated;
  U32                   NumObjects;
} USB_MTP_INIT_DATA;
```

Structure members

Member	Description
EPIn	Endpoint for receiving data from host.
EPOut	Endpoint for sending data to host.
EPInt	Endpoint for sending events to host.
pObjectList	Pointer to a memory region where the list of MTP objects is stored. Should be 4 byte aligned. Each object requires a minimum of 12 bytes + the size of the file name inside the this list. 33 more bytes are needed per object if MTP_SAVE_FILE_INFO is set to 1.
NumBytesObjectList	Number of bytes allocated for the object list.
pDataBuffer	Pointer to a memory region to be used as communication buffer.
NumBytesDataBuffer	Number of bytes allocated for the data buffer.
pMTPInfo	Pointer to a USB_MTP_INFO structure. Filling this structure is mandatory.
InterfaceNum	Internal use.
NumBytesAllocated	Internal use.
NumObjects	Internal use.

Additional information

This structure holds the endpoints that should be used with the MTP interface. Refer to USBD_AddEP() for more information about how to add an endpoint.

The number of bytes in the pDataBuffer should be a multiple of USB maximum packet size. The number of bytes in the object list depends on the number of files/directories on the storage medium. An object is assigned to each file/directory when the USB host requests the object information for the first time.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.2.3 USB_MTP_INFO

Description

Structure that is used when initialising the MTP module.

Type definition

```
typedef struct USB_MTP_INFO {
  const char  * pManufacturer;
  const char  * pModel;
  const char  * pDeviceVersion;
  const char  * pSerialNumber;
} USB_MTP_INFO;
```

Structure members

Member	Description
pManufacturer	Name of the device manufacturer.
pModel	Model name of the MTP device.
pDeviceVersion	Version of the MTP device.
pSerialNumber	Serial number of the MTP device. The serial number should contain exactly 32 hexadecimal characters. It must be unique among devices sharing the same model name and device version strings. The MTP device returns this string in the Serial Number field of the DeviceInfo dataset. For more information, refer to MTP specification.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.2.4 USB_MTP_INST_DATA

Description

Structure which stores the parameters of storage driver.

Type definition

```
typedef struct USB_MTP_INST_DATA {
  const USB_MTP_STORAGE_API  * pAPI;
  const char                 * sDescription;
  const char                 * sVolumeId;
  USB_MTP_INST_DATA_DRIVER     DriverData;
} USB_MTP_INST_DATA;
```

Structure members

Member	Description
pAPI	Pointer to a structure that holds the storage device driver API.
sDescription	Human-readable string which identifies the storage. This string is displayed in Nautilus/Windows Explorer/etc.
sVolumeId	Unique volume identifier This field must be up to 256 characters long but only the first 128 are significant and these must be unique for all storages of an MTP device.
DriverData	Driver data that are passed to the storage driver. Refer to USB_MTP_INST_DATA_DRIVER for detailed information about how to initialize this structure.

Additional information

The MTP device returns the sDescription string in the Storage Description parameter and the sVolumeId in the Volume Identifier of the StorageInfo dataset. For more information, refer to MTP specification.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.2.5 USB_MTP_INST_DATA_DRIVER

Description

Structure which stores the parameters passed to the storage driver.

Type definition

```
typedef struct USB_MTP_INST_DATA_DRIVER {
  const char  * pRootDir;
  U8            IsRemovable;
  U8            Protect;
} USB_MTP_INST_DATA_DRIVER;
```

Structure members

Member	Description
pRootDir	Path to directory to be used as the root of the storage.
IsRemovable	Set to 1 if the storage is removable.
Protect	Set write protection for this storage to prevent writing / deleting of files or formating the storage, see MTP_STORAGE_PROTECT_... macros.

Additional information

pRootDir can specify the root of the file system or any other subdirectory.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.2.6 USB_MTP_STORAGE_API

Description

Structure that contains callbacks to the storage driver.

Type definition

```
typedef struct USB_MTP_STORAGE_API {
  USB_MTP_STORAGE_INIT                    * pfInit;
  USB_MTP_STORAGE_GET_INFO                * pfGetInfo;
  USB_MTP_STORAGE_FIND_FIRST_FILE         * pfFindFirstFile;
  USB_MTP_STORAGE_FIND_NEXT_FILE          * pfFindNextFile;
  USB_MTP_STORAGE_OPEN_FILE               * pfOpenFile;
  USB_MTP_STORAGE_CREATE_FILE             * pfCreateFile;
  USB_MTP_STORAGE_READ_FROM_FILE          * pfReadFromFile;
  USB_MTP_STORAGE_WRITE_TO_FILE           * pfWriteToFile;
  USB_MTP_STORAGE_CLOSE_FILE              * pfCloseFile;
  USB_MTP_STORAGE_REMOVE_FILE             * pfRemoveFile;
  USB_MTP_STORAGE_CREATE_DIR              * pfCreateDir;
  USB_MTP_STORAGE_REMOVE_DIR              * pfRemoveDir;
  USB_MTP_STORAGE_FORMAT                  * pfFormat;
  USB_MTP_STORAGE_RENAME_FILE             * pfRenameFile;
  USB_MTP_STORAGE_DEINIT                  * pfDeInit;
  USB_MTP_STORAGE_GET_FILE_ATTRIBUTES     * pfGetFileAttributes;
  USB_MTP_STORAGE_MODIFY_FILE_ATTRIBUTES  * pfModifyFileAttributes;
  USB_MTP_STORAGE_GET_FILE_CREATION_TIME  * pfGetFileCreationTime;
  USB_MTP_STORAGE_GET_FILELAST_WRITE_TIME * pfGetFileLastWriteTime;
  USB_MTP_STORAGE_GET_FILE_ID             * pfGetFileId;
  USB_MTP_STORAGE_GET_FILE_SIZE           * pfGetFileSize;
  USB_MTP_STORAGE_GET_FILE_INFO           * pfGetFileInfo;
} USB_MTP_STORAGE_API;
```

Structure members

Member	Description
pfInit	Initializes the storage medium.
pfGetInfo	Returns information about the storage medium such as storage capacity and the available free space.
pfFindFirstFile	Returns information about the first file in a given directory.
pfFindNextFile	Moves to next file and returns information about it.
pfOpenFile	Opens an existing file.
pfCreateFile	Creates a new file.
pfReadFromFile	Reads data from the current file.
pfWriteToFile	Writes data to current file.
pfCloseFile	Closes the current file.
pfRemoveFile	Removes a file from storage medium.
pfCreateDir	Creates a new directory.
pfRemoveDir	Removes a directory from storage medium.
pfFormat	Formats the storage.
pfRenameFile	Changes the name of a file or directory.
pfDeInit	De-initializes the storage medium.
pfGetFileAttributes	Reads the attributes of a file or directory.
pfModifyFileAttributes	Changes the attributes of a file or directory.
pfGetFileCreationTime	Returns the creation time of a file or directory.

 © 2010-2026 SEGGER Microcontroller GmbH

Member	Description
pfGetFileLastWrite-Time	Returns the time of the last modification made to a file or directory.
pfGetFileId	Returns the unique ID of a file or directory.
pfGetFileSize	Returns the size of a file in bytes.
pfGetFileInfo	[Optional] Returns information about a file.

Additional information

USB_MTP_STORAGE_API is used to retrieve information from the storage device driver or access data that needs to be read or written. Detailed information can be found in *MTP Storage Driver* on page 353.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.2.7 USB_MTP_STORAGE_INFO

Description

Structure which stores information about a storage.

Type definition

```
typedef struct USB_MTP_STORAGE_INFO {
  U32    NumKBytesTotal;
  U32    NumKBytesFreeSpace;
  U16    FSType;
  U8     IsWriteProtected;
  U8     IsRemovable;
  char   DirDelimiter;
  U8     BigFileSupport;
} USB_MTP_STORAGE_INFO;
```

Structure members

Member	Description
NumKBytesTotal	Storage capacity in kBytes
NumKBytesFreeSpace	Available free space on storage in kBytes
FSType	Type of file system as specified by MTP
IsWriteProtected	Set to 1 if the storage medium can not be modified
IsRemovable	Set to 1 if the storage medium can be removed from device
DirDelimiter	Character which separates the directory/file names in a path
BigFileSupport	Store layer should set this to 1 if it supports files > 4GB.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.2.8 USB_MTP_OPERATION_INFO

Description

Structure which provides information about a new MTP operation.

Type definition

```
typedef struct USB_MTP_OPERATION_INFO {
  const char  * pFilePath;
  U8            IsDirectory;
} USB_MTP_OPERATION_INFO;
```

Structure members

Member	Description
pFilePath	Full path to file.
IsDirectory	Set to 1 if the path points to a directory.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.3 Defines

9.4.3.1 Write protection attributes

Description

Flags to set write protection for a storage. To be used in the USB_MTP_INST_DATA_DRIVER structure.

Definition

```
#define MTP_STORAGE_PROTECT_WRITE     0x01u
#define MTP_STORAGE_PROTECT_DELETE    0x02u
#define MTP_STORAGE_PROTECT_FORMAT    0x04u
#define MTP_STORAGE_PROTECT_ALL       0x07u
```

Symbols

Definition	Description
MTP_STORAGE_PROTECT_WRITE	Prohibit file write
MTP_STORAGE_PROTECT_DELETE	Prohibit file delete
MTP_STORAGE_PROTECT_FORMAT	Prohibit storage formatting
MTP_STORAGE_PROTECT_ALL	Prohibit all write access

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.4 Enums

9.4.4.1 USB_MTP_EVENT

Description

Enum containing the MTP event codes.

Type definition

```
typedef enum {
  USB_MTP_EVENT_UNDEFINED,
  USB_MTP_EVENT_CANCELTRANSACTION,
  USB_MTP_EVENT_OBJECTADDED,
  USB_MTP_EVENT_OBJECTREMOVED,
  USB_MTP_EVENT_STOREADDED,
  USB_MTP_EVENT_STOREREMOVED,
  USB_MTP_EVENT_DEVICEPROPCHANGED,
  USB_MTP_EVENT_OBJECTINFOCHANGED,
  USB_MTP_EVENT_DEVICEINFOCHANGED,
  USB_MTP_EVENT_REQUESTOBJECTTRANSFER,
  USB_MTP_EVENT_STOREFULL,
  USB_MTP_EVENT_DEVICERESET,
  USB_MTP_EVENT_STORAGEINFOCHANGED,
  USB_MTP_EVENT_CAPTURECOMPLETE,
  USB_MTP_EVENT_UNREPORTEDSTATUS,
  USB_MTP_EVENT_OBJECTPROPCHANGED,
  USB_MTP_EVENT_OBJECTPROPDESCCHANGED,
  USB_MTP_EVENT_OBJECTREFERENCESCHANGED
} USB_MTP_EVENT;
```

Enumeration constants

Constant	Description
USB_MTP_EVENT_UNDEFINED	This event code is undefined, and is not used
USB_MTP_EVENT_CANCELTRANSACTION	This event is used to initiate the cancellation of a transaction over transports which do not have their own mechanism for canceling transactions. Currently not used.
USB_MTP_EVENT_OBJECTADDED	This event informs the host about a new object that has been added to the storage.
USB_MTP_EVENT_OBJECTREMOVED	Informs the host that an object has been removed.
USB_MTP_EVENT_STOREADDED	This event indicates that a storage has been added to the device. It allows to dynamically show the available storages.
USB_MTP_EVENT_STOREREMOVED	This event indicates that a storage has been removed to the device. It allows to dynamically hide the available storages.
USB_MTP_EVENT_DEVICEPROPCHANGED	A property changed on the device has occurred. Currently not used.
USB_MTP_EVENT_OBJECTINFOCHANGED	This event indicates that the information for a particular object has changed and that the host should acquire the information once again.

 © 2010-2026 SEGGER Microcontroller GmbH

Constant	Description
USB_MTP_EVENT_DEVICEINFOCHANGED	This event indicates that the capabilities of the device have changed and that the DeviceInfo should be requested again. Currently not used.
USB_MTP_EVENT_REQUESTOBJECTTRANSFER	This event can be used by the device to ask the host to initiate a file object transfer to it. Currently not used.
USB_MTP_EVENT_STOREFULL	This event should be sent when a storage becomes full.
USB_MTP_EVENT_DEVICERESET	Notifies the host about an internal reset. Currently not used
USB_MTP_EVENT_STORAGEINFOCHANGED	This event is used when information of a storage changes.
USB_MTP_EVENT_CAPTURECOMPLETE	Informs the host that the previously initiated capture acquire is complete. Currently not used.
USB_MTP_EVENT_UNREPORTEDSTATUS	This event may be implemented for certain transports in cases where the responder unable to report events to the initiator regarding changes in its internal status. Currently not used.
USB_MTP_EVENT_OBJECTPROPCHANGED	Informs about a change in the object property of an specific object. Currently not used.
USB_MTP_EVENT_OBJECTPROPDESCCHANGED	This event informs that the property description of an object property has been changed and needs to be re-acquired. Currently not used.
USB_MTP_EVENT_OBJECTREFERENCESCHANGED	This event is used to indicate that the references on an object have been updated. Currently not used.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.4.2 USB_MTP_OPERATION_CB_TYPE

Description

Enum containing the callback operation types.

Type definition

```
typedef enum {
  USB_MTP_OPERATION_OBJECT_ADDED,
  USB_MTP_OPERATION_OBJECT_REMOVED,
  USB_MTP_OPERATION_OBJECT_RENAMED_OLD_NAME,
  USB_MTP_OPERATION_OBJECT_RENAMED_NEW_NAME
} USB_MTP_OPERATION_CB_TYPE;
```

Enumeration constants

Constant	Description
USB_MTP_OPERATION_OBJECT_ADDED	A new object has been added.
USB_MTP_OPERATION_OBJECT_REMOVED	An object is about to be removed.
USB_MTP_OPERATION_OBJECT_RENAMED_OLD_NAME	An object is being renamed - old name of the object.
USB_MTP_OPERATION_OBJECT_RENAMED_NEW_NAME	An object is being renamed - new name of the object.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.4.3 USB_MTP_STORAGE_CB_TYPE

Description

Enum containing the callback operation types for the USB_MTP_WP_CB callback registered via USBD_MTP_SetWriteProtectionCb()

Type definition

```
typedef enum {
  USB_MTP_STORAGE_CB_FILE_CREATE,
  USB_MTP_STORAGE_CB_FILE_REMOVE,
  USB_MTP_STORAGE_CB_DIR_CREATE,
  USB_MTP_STORAGE_CB_DIR_REMOVE,
  USB_MTP_STORAGE_CB_RENAME,
  USB_MTP_STORAGE_CB_MODIFY
} USB_MTP_STORAGE_CB_TYPE;
```

Enumeration constants

Constant	Description
USB_MTP_STORAGE_CB_FILE_CREATE	File is being created.
USB_MTP_STORAGE_CB_FILE_REMOVE	File is being deleted.
USB_MTP_STORAGE_CB_DIR_CREATE	Directory is being created.
USB_MTP_STORAGE_CB_DIR_REMOVE	Directory is being removed.
USB_MTP_STORAGE_CB_RENAME	File or dir is being renamed.
USB_MTP_STORAGE_CB_MODIFY	File or dir attributes are being modified.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.5 Prototypes

9.4.5.1 USB_MTP_OBJECT_ALLOC_FAIL

Description

Callback which can be set via `USBD_MTP_SetObjectAllocFailCb()`. This callback is called when the object list runs out of memory for new objects. It can be used to notify the user of the issue (e.g. set an error LED).

Type definition

```
typedef void (USB_MTP_OBJECT_ALLOC_FAIL)(      U32    NumBytesRequested,
                                                U32    NumBytesAvail,
                                          const char * pFilePath,
                                          const char * pFileName);
```

Parameters

Parameter	Description
`NumBytesRequested`	Bytes need for a new object.
`NumBytesAvail`	Bytes free in the object list.
`pFilePath`	Pointer to a string containing the file path.
`pFileName`	Pointer to a string containing the file name.

Additional information

This callback is informative only, the application must not try to free the object list. This callback is called for every object where allocation failed. The callback may not block. When this callback is set the behavior of the MTP module is changed slightly - new objects are normally allocated for each file/dir in a directory which is opened by the user in the PC's explorer. When this callback is not set once a single allocation fails the module will return an error to the PC even if some objects inside a folder could be allocated. When this callback is set the module will return as many objects as it could fit into the object list before allocating started failing (e.g. if a folder contains 50 files and allocation starts failing after 40 files the MTP module will return the first 40 objects to the PC).

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.5.2 USB_MTP_OPERATION_CB

Description

Callback which can be set via USBD_MTP_SetOperationCb(). This callback is called when operations are executed by the host operating system via MTP. This can be used to e.g. monitor new objects being created.

Type definition

```
typedef void (USB_MTP_OPERATION_CB)(      USB_MTP_OPERATION_CB_TYPE OperationType,
                                    const USB_MTP_OPERATION_INFO  * pOpInfo);
```

Parameters

Parameter	Description
OperationType	One of the USB_MTP_OPERATION_CB_TYPE enum values.
pFileInfo	Pointer to a USB_MTP_OPERATION_INFO structure containing information about the affected file.

 © 2010-2026 SEGGER Microcontroller GmbH

9.4.5.3 USB_MTP_WP_CB

Description

Callback which can be set via USBD_MTP_SetWriteProtectionCb(). This optional callback will be called on all write operations (write, rename, create, delete, modify) and allows the application to prevent the operation for selected files. E.g. prevent a certain file from being deleted or modified by the host.

Type definition

```
typedef int (USB_MTP_WP_CB)(      USB_MTP_STORAGE_CB_TYPE OperationType,
                                  U8                      Unit,
                            const char                  * pPath);
```

Parameters

Parameter	Description
OperationType	One of the USB_MTP_STORAGE_CB_TYPE enum values.
Unit	Logical unit number. Specifies for which storage medium the function is called.
pPath	Pointer to a string containing the full file/directory path.

Return value

= 0 Proceed with operation.
≠ 0 Prevent operation, an error will be sent to the host.

Additional information

The callback may not block. It is advised to use MTP_STORAGE_PROTECT_FORMAT for the volume on which the file is located, otherwise it can be deleted if the user formats the volume.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5 MTP Storage Driver

This section describes the emUSB-Device MTP storage interface in detail.

9.5.1 General information

This release comes with the USB_MTP_StorageFS driver, which uses emFile to access the storage medium. If you are using emFile, this chapter can be ignored. This chapter is for those who wish to write a file system interface for a third-party file system.

The storage interface is handled through an API-table, which contains all relevant functions necessary for read/write operations and initialization. Its implementation handles the details of how data is actually read from or written to memory.

9.5.2 Interface function list

As described above, access to a storage medium is realized through an API-function table of type USB_MTP_STORAGE_API. The structure is declared in USB_MTP.h, and it is described in section on page 336.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3 USB_MTP_STORAGE_API in detail

9.5.3.1 USB_MTP_STORAGE_INIT

Description

Initializes the storage medium.

Type definition

```
typedef void (USB_MTP_STORAGE_INIT)(      U8                          Unit,
                                    const USB_MTP_INST_DATA_DRIVER * pDriverData);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Pointer to a USB_MTP_INST_DATA_DRIVER structure that contains all information that is necessary for the driver initialization. For detailed information about the USB_MTP_INST_DATA_DRIVER structure, refer to USB_MTP_INST_DATA_DRIVER.

Additional information

This function is called when the storage driver is added to emUSB-Device-MTP. It is the first function of the storage driver to be called.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.2 USB_MTP_STORAGE_GET_INFO

Description

Returns information about storage medium such as capacity and available free space.

Type definition

```
typedef void (USB_MTP_STORAGE_GET_INFO)(U8                      Unit,
                                        USB_MTP_STORAGE_INFO * pStorageInfo);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pStorageInfo	Pointer to a USB_MTP_STORAGE_INFO structure. For detailed information about the USB_MTP_STORAGE_INFO structure, refer to USB_MTP_STORAGE_INFO.

Additional information

Typically, this function is called immediately after the device is connected to USB host when the USB host requests information about the available storage mediums.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.3 USB_MTP_STORAGE_FIND_FIRST_FILE

Description

Returns information about the first file in a specified directory.

Type definition

```
typedef int (USB_MTP_STORAGE_FIND_FIRST_FILE)(      U8                   Unit,
                                              const char               * pDirPath,
                                                    USB_MTP_FILE_INFO  * pFileInfo);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pDirPath	Full path to the directory to be searched.
pFileInfo	out Information about the file/directory found.

Return value

= 0 File/directory found
= 1 No more files/directories found
< 0 An error occurred

Additional information

The “.” and “..” directory entries which are relevant only for the file system should be skipped.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.4 USB_MTP_STORAGE_FIND_NEXT_FILE

Description

Moves to next file and returns information about it.

Type definition

```
typedef int (USB_MTP_STORAGE_FIND_NEXT_FILE)(U8                  Unit,
                                              USB_MTP_FILE_INFO * pFileInfo);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFileInfo	out Information about the file/directory found.

Return value

= 0 File/directory found
= 1 No more files/directories found
< 0 An error occurred

Additional information

The “.” and “..” directory entries which are relevant only for the file system should be skipped.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.5 USB_MTP_STORAGE_OPEN_FILE

Description

Opens a file for reading.

Type definition

```
typedef int (USB_MTP_STORAGE_OPEN_FILE)(      U8     Unit,
                                        const char * pFilePath);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	in Full path to file.

Return value

= 0 File opened
≠ 0 An error occurred

Additional information

This function is called at the beginning of a file read operation. It is followed by one or more calls to USB_MTP_STORAGE_READ_FROM_FILE. At the end of data transfer the MTP module closes the file by calling USB_MTP_STORAGE_CLOSE_FILE. If the file does not exists an error should be returned. The MTP module opens only one file at a time.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.6 USB_MTP_STORAGE_CREATE_FILE

Description

Opens a file for writing.

Type definition

```
typedef int (USB_MTP_STORAGE_CREATE_FILE)(      U8                   Unit,
                                          const char               * pDirPath,
                                                USB_MTP_FILE_INFO  * pFileInfo);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pDirPath	in Full path to directory where the file should be created.
pFileInfo	in Information about the file to be created. pFileName points to the name of the file. out pFilePath points to full path of created file, pFileName points to the beginning of file name in pFilePath.

Return value

= 0 File created and opened
≠ 0 An error occurred

Additional information

This function is called at the beginning of a file write operation. The name of the file is specified in the pFileName filed of pFileInfo. If the file exists it should be truncated to zero length. When a file is created, the call to USB_MTP_STORAGE_CREATE_FILE is followed by one or more calls to USB_MTP_STORAGE_WRITE_TO_FILE. If CreationTime and LastWriteTime in pFileInfo are not zero, these should be used instead of the time stamps generated by the file system.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.7 USB_MTP_STORAGE_READ_FROM_FILE

Description

Reads data from the current file.

Type definition

```
typedef int (USB_MTP_STORAGE_READ_FROM_FILE)(U8     Unit,
                                             U64    Off,
                                             void * pData,
                                             U32    NumBytes);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
Off	Byte offset where to read from.
pData	out Data read from file.
NumBytes	Number of bytes to read from file.

Return value

= 0 Data read from file
≠ 0 An error occurred

Additional information

The function reads data from the file opened by USB_MTP_STORAGE_OPEN_FILE.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.8 USB_MTP_STORAGE_WRITE_TO_FILE

Description

Writes data to current file.

Type definition

```
typedef int (USB_MTP_STORAGE_WRITE_TO_FILE)(      U8     Unit,
                                                  U64    Off,
                                            const void * pData,
                                                  U32    NumBytes);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
Off	Byte offset where to read from.
pData	in Data to write to file.
NumBytes	Number of bytes to write to file.

Return value

= 0 Data written to file
≠ 0 An error occurred

Additional information

The function writes data to file opened by USB_MTP_STORAGE_CREATE_FILE.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.9 USB_MTP_STORAGE_CLOSE_FILE

Description

Closes the current file.

Type definition

```
typedef int (USB_MTP_STORAGE_CLOSE_FILE)(U8 Unit);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.

Return value

= 0 File closed.
≠ 0 An error occurred

Additional information

The function closes the file opened by USB_MTP_STORAGE_CREATE_FILE or USB_MTP_STORAGE_OPEN_FILE.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.10 USB_MTP_STORAGE_REMOVE_FILE

Description

Removes a file/directory from the storage medium.

Type definition

```
typedef int (USB_MTP_STORAGE_REMOVE_FILE)(      U8     Unit,
                                          const char * pFilePath);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file/directory to be removed

Return value

= 0 File removed.
≠ 0 An error occurred

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.11 USB_MTP_STORAGE_CREATE_DIR

Description

Creates a directory on the storage medium.

Type definition

```
typedef int (USB_MTP_STORAGE_CREATE_DIR)(      U8                  Unit,
                                         const char              * pDirPath,
                                               USB_MTP_FILE_INFO * pFileInfo);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pDirPath	in Full path to directory where the directory should be created.
pFileInfo	in Information about the directory to be created. pFileName points to the directory name. out pFilePath points to full path of directory, pFileName points to the beginning of directory name in pFilePath

Return value

= 0 Directory created.
≠ 0 An error occurred

Additional information

If CreationTime and LastWriteTime in pFileInfo are not available, zero should be used instead of the time stamps generated by the file system.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.12 USB_MTP_STORAGE_REMOVE_DIR

Description

Removes a directory and its contents from the storage medium.

Type definition

```
typedef int (USB_MTP_STORAGE_REMOVE_DIR)(      U8     Unit,
                                         const char * pDirPath);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pDirPath	in Full path to directory to be removed.

Return value

= 0 Directory removed.
≠ 0 An error occurred

Additional information

The function should remove the directory and the entire file tree under it.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.13 USB_MTP_STORAGE_FORMAT

Description

Initializes the storage medium.

Type definition

```
typedef int (USB_MTP_STORAGE_FORMAT)(U8 Unit);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.

Return value

= 0 Storage medium initialized.
≠ 0 An error occurred

Additional information

The file system layer has to differentiate between two cases, one where the MTP root directory is the same as the root directory of the file system and one where it is only a subdirectory of the file system. If `pRootDir` which was configured in the call to `USB_MTP_STORAGE_INIT`, points to a subdirectory of the file system, the storage medium should not be formatted. Instead, all the files and directories underneath `pRootDir` should be removed.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.14 USB_MTP_STORAGE_RENAME_FILE

Description

Changes the name of a file or directory.

Type definition

```
typedef int (USB_MTP_STORAGE_RENAME_FILE)(U8                  Unit,
                                          USB_MTP_FILE_INFO * pFileInfo);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFileInfo	Pointer to a USB_MTP_FILE_INFO structure. in Information about the file/directory to be renamed. pFilePath member points to the full path and pFileName points to the new name. out pFilePath member points to full path of file/directory with the new name, pFileName points to the beginning of file/directory name in pFilePath. The other structure fields should also be filled.

Return value

= 0 File/directory renamed.
≠ 0 An error occurred

Additional information

Only the name of the file/directory should be changed. The path to parent directory should remain the same.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.15 USB_MTP_STORAGE_DEINIT

Description

De-initializes the storage medium.

Type definition

```
typedef void (USB_MTP_STORAGE_DEINIT)(U8 Unit);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.

Additional information

Typically called when the application calls `USBD_Stop()` to de-initialize emUSB-Device.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.16 USB_MTP_STORAGE_GET_FILE_ATTRIBUTES

Description

Returns the attributes of a file or directory.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILE_ATTRIBUTES)(      U8     Unit,
                                                  const char * pFilePath,
                                                        U8   * pMask);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
pMask	out The bitmask of the attributes.

Return value

= 0 Information returned.
≠ 0 An error occurred

Additional information

This function is called only when the compile time switch MTP_SAVE_FILE_INFO is set to 0. For the list of supported attributes refer to USB_MTP_FILE_INFO.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.17 USB_MTP_STORAGE_MODIFY_FILE_ATTRIBUTES

Description

Sets and clears file attributes.

Type definition

```
typedef int (USB_MTP_STORAGE_MODIFY_FILE_ATTRIBUTES)(      U8     Unit,
                                                     const char * pFilePath,
                                                           U8     SetMask,
                                                           U8     ClrMask);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
SetMask	The bitmask of the attributes which should be set.
ClrMask	The bitmask of the attributes which should be cleared.

Return value

= 0 Attributes modified.
≠ 0 An error occurred

Additional information

For the list of supported attributes refer to USB_MTP_FILE_INFO.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.18 USB_MTP_STORAGE_GET_FILE_CREATION_TIME

Description

Returns the creation time of file or directory.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILE_CREATION_TIME)(      U8     Unit,
                                                     const char * pFilePath,
                                                           U32  * pTime);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
pTime	out The creation time.

Return value

= 0 Creation time returned.
≠ 0 An error occurred

Additional information

This function is called only when the compile time switch MTP_SAVE_FILE_INFO is set to 0. For the list of supported attributes refer to USB_MTP_FILE_INFO.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.19 USB_MTP_STORAGE_GET_FILELAST_WRITE_TIME

Description

Returns the time when the file or directory was last modified.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILELAST_WRITE_TIME)(      U8     Unit,
                                                      const char * pFilePath,
                                                            U32  * pTime);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
pTime	out The modification time.

Return value

= 0 Modification time returned.
≠ 0 An error occurred

Additional information

This function is called only when the compile time switch MTP_SAVE_FILE_INFO is set to 0. For the list of supported attributes refer to USB_MTP_FILE_INFO.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.20 USB_MTP_STORAGE_GET_FILE_ID

Description

Returns an ID which uniquely identifies the file or directory.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILE_ID)(      U8     Unit,
                                          const char * pFilePath,
                                                U8   * pId);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
pId	out The unique ID of file or directory. Should point to a byte array MTP_NUM_BYTES_FILE_ID large.

Return value

= 0 ID returned.
≠ 0 An error occurred

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.21 USB_MTP_STORAGE_GET_FILE_SIZE

Description

Returns the size of a file in bytes.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILE_SIZE)(      U8     Unit,
                                            const char * pFilePath,
                                                  U64  * pFileSize);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Full path to file or directory (0-terminated string).
pFileSize	out The size of file in bytes.

Return value

= 0 Size of file returned.
≠ 0 An error occurred

Additional information

This function is called only when the compile time switch MTP_SAVE_FILE_INFO is set to 0.

 © 2010-2026 SEGGER Microcontroller GmbH

9.5.3.22 USB_MTP_STORAGE_GET_FILE_INFO

Description

This function is optional. It is used to speed up retrieval of file information. Returns the creation time, modification time and attributes in one call.

Type definition

```
typedef int (USB_MTP_STORAGE_GET_FILE_INFO)(      U8     Unit,
                                            const char * pFilePath,
                                                  U32  * pCreationTime,
                                                  U32  * pLastWriteTime,
                                                  U64  * pFileSize,
                                                  U8   * pAttributes);
```

Parameters

Parameter	Description
Unit	Logical unit number. Specifies for which storage medium the function is called.
pFilePath	Path to file or directory.
pCreationTime	out The creation time.
pLastWriteTime	out The modification time.
pAttributes	out The size of file in bytes.

Return value

= 0 Operation retrieved information successfully.
≠ 0 An error occurred

Additional information

This function is called only when the compile time switch MTP_SAVE_FILE_INFO is set to 0.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 10

Communication Device Class (CDC)

This chapter describes how to get emUSB-Device up and running as a CDC device.



 © 2010-2026 SEGGER Microcontroller GmbH

10.1 Overview

The Communication Device Class - Abstract Control Model (CDC-ACM) is an abstract USB class protocol defined by the USB Implementers Forum. For simplicity, CDC-ACM is often referred to as simply CDC. This protocol covers the handling of the following communication flows:

- Virtual COM/Serial interface
- Universal modem device
- ISDN communication
- Ethernet communication

A custom USB driver is not necessary because a kernel mode driver for USB-CDC serial communication is delivered by all major operating systems.

Windows

Starting in Windows 10, such a file is not necessary anymore. A generic inf is provided, handling devices/interfaces with a Device-/InterfaceClass = `0x02` or Device-/InterfaceClass = `0x02` and Device-/InterfaceSubClass = `0x02`. You may need to use an `.inf` file for older Windows versions. These are delivered with emUSB-Device. How to use and modify these files can be found on the SEGGER wiki pages.

Linux

Linux has handled USB 2 virtual COM ports since Kernel Ver. 2.4. Further information can be found in the Linux Kernel documentation.

macOS

macOS (formerly also known as OS X) has supported CDC-ACM devices since the first release, 10.1. The kext that is loaded is called com.apple.driver.usb.cdc.acm.

10.1.1 Configuration

The configuration section should later be modified to match the real application. For the purpose of getting emUSB-Device up and running as well as doing an initial test, the configuration as delivered should not be modified.

10.1.2 CDC-ACM issues on Windows 10

Windows 10 comes with a re-designed driver for CDC-ACM. At the time of writing (June 2019 (re-confirmed in January 2021)), Windows 10 has an issue with large IN CDC transfers. Sometimes packets seem to disappear inside the Windows 10 USB stack. The only workaround is to read in small chunks or to add a delay to the transfers. We have analyzed this using a hardware USB analyzer and a test program which reads data from the device. The device (USB high-speed) sends out data continuously in 512-byte packets. Each packet has a unique, consecutive ID. The test application checks that the received packet always has the ID of the previous packet + 1. After a couple of dozen packets, the error usually appears, and a packet ends up missing. When comparing the packets which are seen "on the wire" using the USB analyzer with the packets which the Windows 10 program received, it can be seen that sometimes packets are missing even though they were clearly successfully received by Windows 10. E.g., on the analyzer one can see packets 30, 31, 32, 33, and on Windows 10 one can see 30, 31, 33. It would appear that the method through which data is read from the COM port (Windows API ReadFile, ReadFile overlapped or ReadFileEx) has no effect on the missing packets. With the latest update on Windows 10 and 11, the latest driver has been confirmed to be reliable for the following scenario: For full-speed devices, Windows would allocate the following: every 16 ms, a read frame of 4096 bytes is triggered, or 3 consecutive read transfers are triggered, where all must be less than 4096 bytes in sum, resulting in a transfer rate of 256 kByte/s. For high-speed devices, the intervals are 16 uFrames, which result in a time frame of 2 ms, giving a speed of max. 2048 kiByte/s. As there is no way of changing this, this is the default behavior of Microsoft's usbser.sys. Refer to {MS Learn Q&A Changing the transfer size and latency for

 © 2010-2026 SEGGER Microcontroller GmbH

usbser driver} or {Serial Port Latency Option Missing - Windows 11}. It is also important to read the data from the target on Windows as fast as possible; otherwise, the above-mentioned issue will happen. When using the same program on Windows 7, no issues can be seen. Another issue that affects all versions of the usbser.sys driver: the RTS setting is only altered on the device when DTR is changed.

 © 2010-2026 SEGGER Microcontroller GmbH

10.2 The example application

The start application (in the `Application` subfolder) is a simple echo server, which can be used to test emUSB-Device. The application receives data byte by byte and sends it back to the host.

Source code excerpt from `USB_CDC_Echo.c`:

```
/*********************************************************************
*
*       MainTask
*/
void MainTask(void);
void MainTask(void) {
  USB_CDC_HANDLE hInst;
  USBD_Init();
  hInst = _AddCDC();
  USBD_SetDeviceInfo(&_DeviceInfo);
  USBD_Start();
  while (1) {
    static char _ac[USB_HS_BULK_MAX_PACKET_SIZE];
    int         NumBytesReceived;

    //
    // Wait for configuration
    //
    while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED)) !
= USB_STAT_CONFIGURED) {
      BSP_ToggleLED(0);
      USB_OS_Delay(50);
    }
    BSP_SetLED(0);
    //
    // Receive at maximum of 64 Bytes
    // If less data has been received,
    // this should be OK.
    //
    NumBytesReceived = USBD_CDC_Receive(hInst, &_ac[0], sizeof(_ac), 0);
    if (NumBytesReceived > 0) {
      USBD_CDC_Write(hInst, &_ac[0], NumBytesReceived, 0);
    }
  }
}
```

10.2.1 Testing communication to the USB device

Accessing CDC on Windows

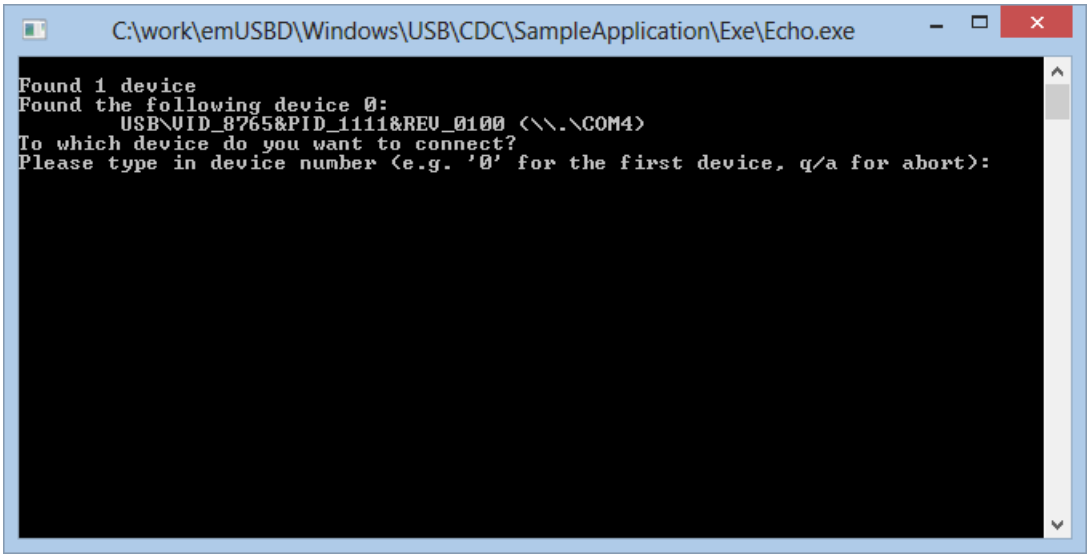
The start application is a simple echo server. This means each character that is entered and sent through the virtual serial port will be sent back by the USB device. A simple Windows sample application is available to test the start application. The application is located in `Windows\USB\CDC\SampleApplication\Exe`.

Alternatively, any terminal program, such as PuTTY or TerraTerm or RealTerm, can be used to check the connectivity.

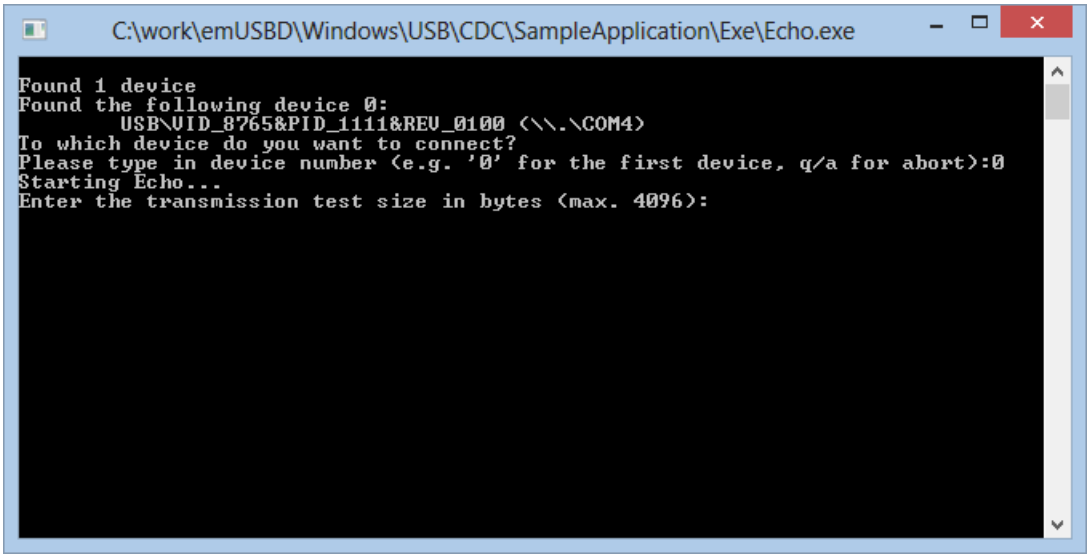
This section shows how to start and run the sample application for the first time.

- Go to the `Windows\USB\CDC\SampleApplication\Exe` folder and double-click on the Echo application. A console window will open and will show that a device has been found with the desired CDC Product and Vendor ID. Enter 0 to connect to that device.

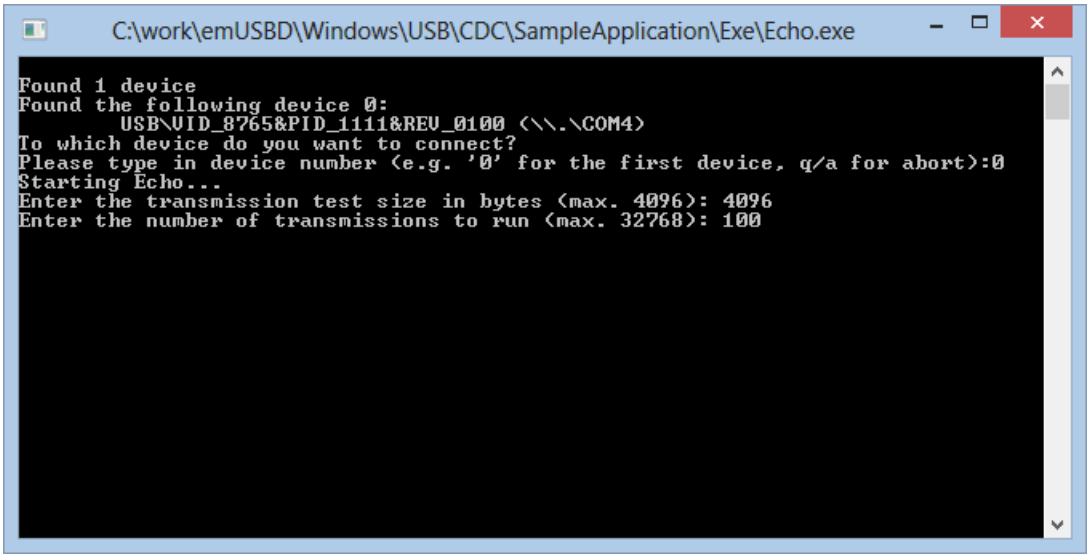
 © 2010-2026 SEGGER Microcontroller GmbH



- The application will ask for the number of bytes to send to and receive from the target device.

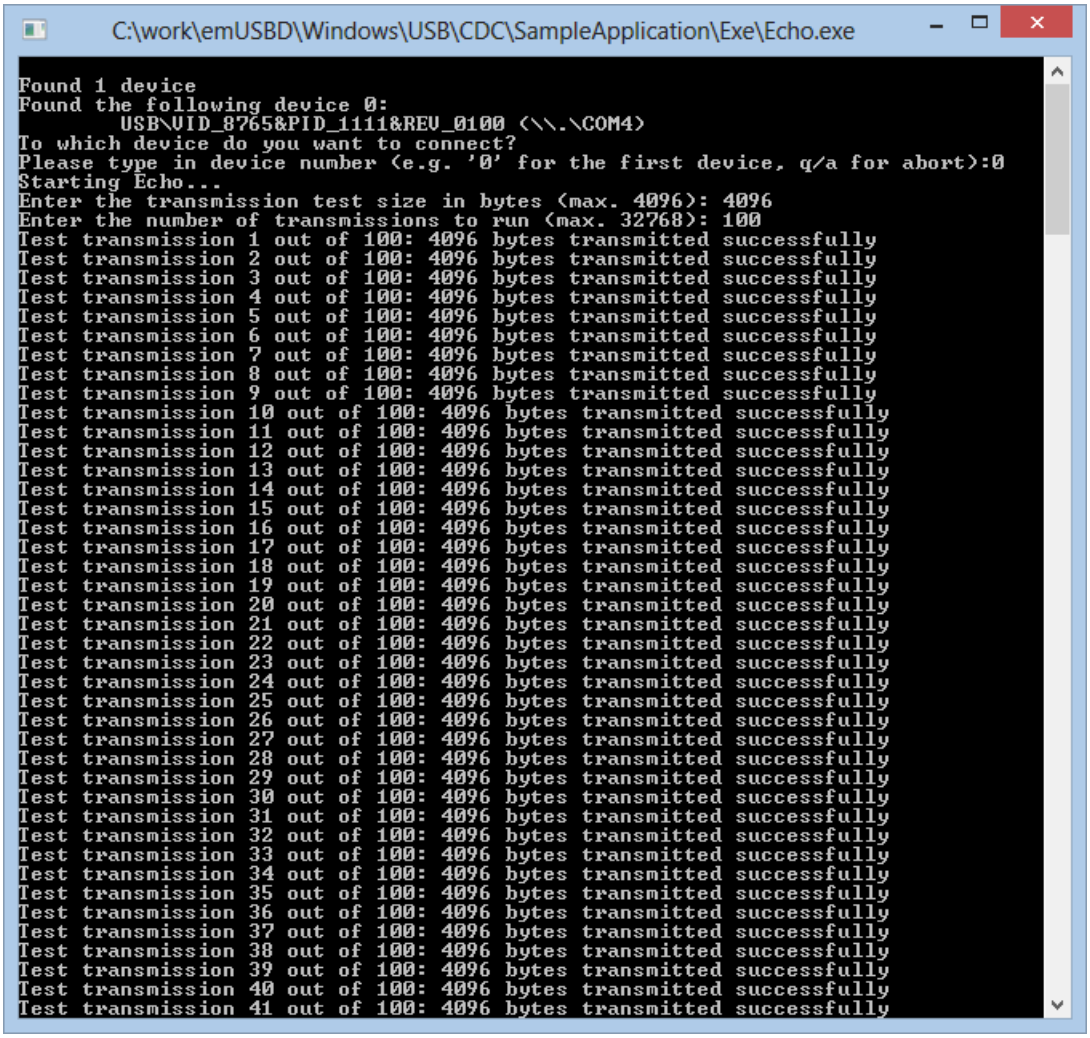


- Now enter the number of repetitions the application shall send and receive to or from the device and confirm with [Enter].



- The test will run and should look like the following screenshot:

 © 2010-2026 SEGGER Microcontroller GmbH



Accessing CDC on Linux

On Linux, no drivers are needed; the device should show up as /dev/ttyACM0 or similar. “sudo screen /dev/ttyACM0 115200” can be used to access the device.

Accessing CDC on macOS

On macOS, no drivers are needed; the device should show up as /dev/tty.usbmodem13245678 or similar. The “screen” terminal program can be used to access the device.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3 Target API

This chapter describes the functions and data structures that can be used with the target application.

10.3.1 Interface function list

Name	Description
API functions	
USBD_CDC_Add()	Adds a CDC-ACM class to the stack.
USBD_CDC_CancelRead()	Cancel a pending read operation.
USBD_CDC_CancelWrite()	Cancel a pending write operation.
USBD_CDC_Read()	Reads data from the host.
USBD_CDC_ReadOverlapped()	Reads data from the host asynchronously.
USBD_CDC_Receive()	Reads data from the host.
USBD_CDC_ReceivePoll()	Reads data from the host.
USBD_CDC_ReadAsync()	Reads data from the host asynchronously.
USBD_CDC_SetOnBreak()	Sets a callback in order to inform the application about a break condition sent by the host.
USBD_CDC_SetOnLineCoding()	Sets a user callback that shall be called when a SET_LINE_CODING request is sent to the device.
USBD_CDC_SetOnControlLineState()	Sets a user callback that shall be called when a SET_LINE_STATE request is sent to the device.
USBD_CDC_SetOnRXEvent()	Sets a callback function for the OUT endpoint that will be called on every RX event for that endpoint.
USBD_CDC_SetOnTXEvent()	Sets a callback function for the IN endpoint that will be called on every TX event for that endpoint.
USBD_CDC_UpdateSerialState()	Sets the new serial state.
USBD_CDC_Write()	Writes data to the host.
USBD_CDC_WriteAsync()	Sends data to the host asynchronously.
USBD_CDC_WaitForRX()	This function is to be used in combination with USBD_CDC_ReadOverlapped() and waits for the reading data transfer from the host to complete.
USBD_CDC_PollForRX()	This function is to be used in combination with USBD_CDC_ReadOverlapped() and waits for the reading data transfer from the host to complete.
USBD_CDC_WaitForTX()	This function is to be used in combination with a non-blocking call to USBD_CDC_Write().
USBD_CDC_PollForTX()	This function is to be used in combination with a non-blocking call to USBD_CDC_Write().
USBD_CDC_WaitForTXReady()	Waits (blocking) until the TX queue can accept another data packet.

 © 2010-2026 SEGGER Microcontroller GmbH

Name	Description
USBD_CDC_WriteSerialState()	Sends the current control line serial state to the host.
USBD_CDC_GetNumBytesRemToRead()	This function is to be used in combination with USBD_CDC_ReadOverlapped().
USBD_CDC_GetNumBytesRemToWrite()	This function is to be used in combination with a non-blocking call to USBD_CDC_Write().
USBD_CDC_GetNumBytesInBuffer()	Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer.
Data structures	
USB_CDC_INIT_DATA	Initialization structure that is needed when adding a CDC interface to emUSB-Device.
USB_CDC_LINE_CODING	Structure that contains the new line-coding information sent by the host.
USB_CDC_SERIAL_STATE	Structure that contains the serial state that can be sent to the host.
USB_CDC_CONTROL_LINE_STATE	Structure that contains the new control line state sent by the host.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.1 USBD_CDC_Add()

Description

Adds a CDC-ACM class to the stack.

Prototype

```
USB_CDC_HANDLE USBD_CDC_Add(const USB_CDC_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a USB_CDC_INIT_DATA structure.

Return value

Handle to a valid CDC instance. The handle of the first CDC instance is always 0.

Additional information

After the initialization of emUSB-Device, this is the first function that needs to be called when the USB-CDC interface is used with emUSB-Device. The returned value can be used with the CDC functions in order to talk to the right CDC instance.

For creating more than one CDC instance please make sure the USBD_EnableIAD() is called before, otherwise none but the first CDC instance will work correctly. The same is true for composite devices including CDC and another, different USB class.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.2 USBD_CDC_CancelRead()

Description

Cancel a pending read operation.

Prototype

```
void USBD_CDC_CancelRead(USB_CDC_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add-d().

Additional information

This function can be called when a pending asynchronous read operation (triggered by USBD_CDC_ReadOverlapped()) should be canceled. The function can be called from any task.

The function can also be used to cancel a call to one of the blocking read functions (when called from a different task or interrupt function).

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.3 USBD_CDC_CancelWrite()

Description

Cancel a pending write operation.

Prototype

```
void USBD_CDC_CancelWrite(USB_CDC_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().

Additional information

This function shall be called when a pending asynchronous write operation (triggered by non-blocking call to USBD_CDC_Write()) should be canceled. It can be called from any task.

The function can also be used to cancel a call to a blocking write functions (when called from a different task or interrupt function).

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.4 USBD_CDC_Read()

Description

Reads data from the host.

Prototype

```
int USBD_CDC_Read(USB_CDC_HANDLE   hInst,
                  void           * pData,
                  unsigned         NumBytes,
                  unsigned         Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
Timeout	Timeout given in milliseconds. A zero value results in an infinite timeout.

Return value

= NumBytes	Requested data was successfully read within the given timeout.
≥ 0 && < NumBytes	Timeout has occurred (Number of bytes read before timeout).
< 0	An error occurred.

Additional information

This function blocks the task until all data has been read or a timeout occurs. In case of a reset or a disconnect USB_STATUS_ERROR is returned.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via USBD_AddEP(). This data can be retrieved by a later call to USBD_CDC_Receive() / USBD_CDC_Read(). See also USBD_CDC_GetNumBytesInBuffer().

In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.5 USBD_CDC_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

```
int USBD_CDC_ReadOverlapped(USB_CDC_HANDLE   hInst,
                            void           * pData,
                            unsigned         NumBytes);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Return value

≥ 0 Number of bytes that have been read from the internal buffer (success).
= 0 No data was found in the internal buffer (success).
< 0 An error occurred.

Additional information

This function will not block the calling task. The read transfer will be initiated and the function returns immediately. In order to synchronize, USBD_CDC_WaitForRX() needs to be called.

Another synchronization method would be to periodically call USBD_CDC_GetNumBytesRemToRead() in order to see how many bytes still need to be received (this method is preferred when a non-blocking solution is necessary).

The read operation can be canceled using USBD_CDC_CancelRead().

The buffer pointed to by pData must be valid until the read operation is terminated.

Example

See *USBD_CDC_GetNumBytesRemToRead* on page 411.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.6 USBD_CDC_Receive()

Description

Reads data from the host. The function blocks until any data have been received. In contrast to USBD_CDC_Read() this function does not wait for all of NumBytes to be received, but returns after the first packet has been received or after the timeout occurs. In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

Prototype

```
int USBD_CDC_Receive(USB_CDC_HANDLE   hInst,
                     void           * pData,
                     unsigned         NumBytes,
                     int              Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
Timeout	Timeout given in milliseconds. A zero value results in an infinite timeout. If Timeout is -1, the function never blocks and only reads data from the internal endpoint buffer.

Return value

> 0	Number of bytes that have been read within the given timeout.
= 0	A timeout occurred (if Timeout > 0), zero packet received (not every controller supports this!), no data in buffer (if Timeout < 0) or the target was disconnected during the function call and no data was read so far.
< 0	An error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling USBD_CDC_Receive() will return as much data as is currently available up to the size of the buffer specified within the specified timeout. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return the number of bytes read so far. If the target was disconnected before this function was called, it returns USB_STATUS_ERROR.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via USBD_AddEP(). This data can be retrieved by a later call to USBD_CDC_Receive() / USBD_CDC_Read(). See also USBD_CDC_GetNumBytesInBuffer().

A call of USBD_CDC_Receive(Inst, NULL, 0, -1) can be used to trigger an asynchronous read that stores the data into the internal buffer.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.7 USBD_CDC_ReceivePoll()

Description

Reads data from the host. The function blocks until any data have been received. In contrast to USBD_CDC_Read() this function does not wait for all of NumBytes to be received, but returns after the first packet has been received or after the timeout occurs. In contrast to USBD_CDC_Receive() this function will continue the read transfer asynchronously in case of a timeout.

Prototype

```
int USBD_CDC_ReceivePoll(USB_CDC_HANDLE   hInst,
                         void           * pData,
                         unsigned         NumBytes,
                         unsigned         Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
Timeout	Timeout given in milliseconds. A zero value results in an infinite timeout.

Return value

> 0	Number of bytes that have been read within the given timeout.
= 0	A timeout occurred (if Timeout > 0), zero packet received (not every controller supports this!), no data in buffer (if Timeout < 0) or the target was disconnected during the function call and no data was read so far.
< 0	An error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling USBD_CDC_ReceivePoll() will return as much data as is currently available up to the size of the buffer specified within the specified timeout. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return the number of bytes read so far. If the target was disconnected before this function was called, it returns USB_STATUS_ERROR.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via USBD_AddEP(). This data can be retrieved by a later call to USBD_CDC_ReceivePoll() / USBD_CDC_Read(). See also USBD_CDC_GetNumBytesInBuffer().

If a timeout occurs, the read transfer is not affected. Data send from the host after the timeout is stored into the internal buffer of the endpoint and can be read by later calls to USBD_CDC_ReceivePoll().

If Timeout = 0, the function behaves like USBD_CDC_Receive().

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.8 USBD_CDC_ReadAsync()

Description

Reads data from the host asynchronously. The function does not wait for the data to be received. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_CDC_ReadAsync(USB_CDC_HANDLE          hInst,
                        USB_ASYNC_IO_CONTEXT * pContext,
                        int                    ShortRead);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pContext	Pointer to an I/O context containing parameters and pointer to the callback function.
ShortRead	• 0: The transfer is completed successfully after all bytes have been read. • 1: The transfer is completed successfully after one packet has been read.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.9 USBD_CDC_SetOnBreak()

Description

Sets a callback in order to inform the application about a break condition sent by the host.

Prototype

```
void USBD_CDC_SetOnBreak(USB_CDC_HANDLE          hInst,
                         USB_CDC_ON_SET_BREAK * pfOnBreak);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pfOnBreak	Pointer to the callback function.

Additional information

The callback is called in an ISR context, therefore it should execute quickly.

The callback function has the following prototype:

```
typedef void USB_CDC_ON_SET_BREAK(unsigned BreakDuration);
```

Parameter	Description
BreakDuration	Length of the break signal in milliseconds. If BreakDuration is 0xFFFF, this is counted as a permanent break condition. A SendBreak request with BreakDuration of 0x0000 will reset the break state.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.10 USBD_CDC_SetOnLineCoding()

Description

Sets a user callback that shall be called when a SET_LINE_CODING request is sent to the device.

Prototype

```
void USBD_CDC_SetOnLineCoding(USB_CDC_HANDLE                hInst,
                              USB_CDC_ON_SET_LINE_CODING * pf);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pf	Pointer to the callback function.

Additional information

This function is used to register a user callback which notifies the application that the host has changed the line coding.

The callback is called in an ISR context, therefore it should execute quickly.

The callback function has the following prototype:

```
typedef void USB_CDC_ON_SET_LINE_CODING(USB_CDC_LINE_CODING * pLineCoding);
```

Parameter	Description
pLineCoding	Pointer to USB_CDC_LINE_CODING structure containing the new line coding parameters sent from the host.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.11 USBD_CDC_SetOnControlLineState()

Description

Sets a user callback that shall be called when a SET_LINE_STATE request is sent to the device.

Prototype

```
void USBD_CDC_SetOnControlLineState(USB_CDC_HANDLE                      hInst,
                                    USB_CDC_ON_SET_CONTROL_LINE_STATE * pf);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pf	Pointer to the callback function.

Additional information

This function is used to register a user callback which notifies the application that the host has changed the control line state.

The callback is called in an ISR context, therefore it should execute quickly and must never block.

The callback function has the following prototype:

```
typedef void USB_CDC_ON_SET_CONTROL_LINE_STATE(USB_CDC_CONTROL_LINE_STATE * pLineState);
```

Parameter	Description
pLineState	Pointer to USB_CDC_CONTROL_LINE_STATE structure containing the new line state parameters sent from the host.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.12 USBD_CDC_SetOnRXEvent()

Description

Sets a callback function for the OUT endpoint that will be called on every RX event for that endpoint.

Prototype

```
void USBD_CDC_SetOnRXEvent(USB_CDC_HANDLE             hInst,
                           USB_EVENT_CALLBACK       * pEventCb,
                           USB_EVENT_CALLBACK_FUNC  * pfEventCb,
                           void                     * pContext);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure (will be initialized by this function).
pfEventCb	Pointer to the callback routine that will be called on every event on the USB endpoint.
pContext	A pointer which is used as parameter for the callback function.

Additional information

The USB_EVENT_CALLBACK structure is private to the USB stack. It will be initialized by USBD_CDC_SetOnRXEvent(). The USB stack keeps track of all event callback functions using a linked list. The USB_EVENT_CALLBACK structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a read operation was started using one of the USBD_CDC_Read…() or USBD_CDC_Receive() functions.

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
Events	A bit mask indicating which events occurred on the endpoint.
pContext	The pointer which was provided to the USBD_SetOnEvent function.

Note that the callback function will be called within an ISR; therefore, it should never block.

The first parameter to the callback function will contain a bit mask for all events that triggered the call:

Event	Description
USB_EVENT_DATA_READ	Some data was received from the host on the endpoint.
USB_EVENT_READ_COMPLETE	The last read operation was completed.
USB_EVENT_READ_ABORT	A read transfer was aborted.

 © 2010-2026 SEGGER Microcontroller GmbH

Example

```
// The callback function.
static void _OnEvent(unsigned Events, void *pContext) {
  unsigned NumBytes;

  if (Events & USB_EVENT_DATA_READ) {
    NumBytes = USBD_CDC_GetNumBytesInBuffer(hInst);
    if (NumBytes) {
      //
      // The call to receive will read all data from
      // the internal buffer and will start a new transfer.
      // The new transfer will again generate a new event when new data arrives.
      //
      // Note that a new transfer is only started when
      // the internal buffer is completely empty.
      // (It will be empty if you read the number of bytes
      // USBD_CDC_GetNumBytesInBuffer returns.)
      //
      r = USBD_CDC_Receive(hInst, Buff, NumBytes, -1);
      if (r > 0) {
        <.. process data in Buff..>
      }
    }
  }
}
// Main program.
// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBD_CDC_SetOnRXEvent(hInst, &_usb_callback, _OnEvent, NULL);
// Trigger first read
USBD_CDC_Receive(Inst, NULL, 0, -1);
<.. do anything else here while the data is processed in the callback ..>
```

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.13 USBD_CDC_SetOnTXEvent()

Description

Sets a callback function for the IN endpoint that will be called on every TX event for that endpoint.

Prototype

```
void USBD_CDC_SetOnTXEvent(USB_CDC_HANDLE             hInst,
                           USB_EVENT_CALLBACK       * pEventCb,
                           USB_EVENT_CALLBACK_FUNC  * pfEventCb,
                           void                     * pContext);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pEventCb	Pointer to a USB_EVENT_CALLBACK structure (will be initialized by this function).
pfEventCb	Pointer to the callback routine that will be called on every event on the USB endpoint.
pContext	A pointer which is used as parameter for the callback function.

Additional information

The USB_EVENT_CALLBACK structure is private to the USB stack. It will be initialized by USBD_CDC_SetOnTXEvent(). The USB stack keeps track of all event callback functions using a linked list. The USB_EVENT_CALLBACK structure will be included into this linked list and must reside in static memory.

The callback function is called only, if a write operation was started using one of the USBD_CDC_Write…() functions.

The callback function has the following prototype:

```
typedef void USB_EVENT_CALLBACK_FUNC(unsigned Events, void *pContext);
```

Parameter	Description
Events	A bit mask indicating which events occurred on the endpoint.
pContext	The pointer which was provided to the USBD_SetOnEvent function.

Note that the callback function will be called within an ISR, therefore it should never block.

The first parameter to the callback function will contain a bit mask for all events that triggered the call:

Event	Description
USB_EVENT_DATA_SEND	Some data was sent to the host, so that (part of) the user write buffer may be reused by the application.
USB_EVENT_DATA_ACKED	Some data was acknowledged by the host.
USB_EVENT_WRITE_ABORT	A write transfer was aborted.
USB_EVENT_WRITE_COMPLETE	All write operations were completed.

 © 2010-2026 SEGGER Microcontroller GmbH

Example

```
// The callback function.

static void _OnEvent(unsigned Events, void *pContext) {
  if ((Events & USB_EVENT_DATA_SEND) != 0 &&
        // Check for last write transfer to be completed.
        USBD_CDC_GetNumBytesRemToWrite(_hInst) == 0) {
            <.. prepare next data for writing..>
            // Send next packet of data.
            r = USBD_CDC_Write(_hInst, &ac[0], 200, -1);
            if (r < 0) {
              <.. error handling..>
            }
   }
}

// Main program.

// Register callback function.
static USB_EVENT_CALLBACK _usb_callback;
USBD_CDC_SetOnTXEvent(hInst, &_usb_callback, _OnEvent, NULL);
// Send the first packet of data using an asynchronous write operation.
r = USBD_CDC_Write(_hInst, &ac[0], 200, -1);
if (r < 0) {
  <.. error handling..>
}
<.. do anything else here while the whole data is sent..>
```

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.14 USBD_CDC_UpdateSerialState()

Description

Sets the new serial state. This function updates the control line state internally. In order to inform the host about the serial state change, refer to the function `USBD_CDC_WriteSerialState()`.

Prototype

```
void USBD_CDC_UpdateSerialState(      USB_CDC_HANDLE         hInst,
                                const USB_CDC_SERIAL_STATE * pSerialState);
```

Parameters

Parameter	Description
`hInst`	Handle to a valid CDC instance, returned by `USBD_CDC_Add()`.
`pSerialState`	Pointer to the `USB_CDC_SERIAL_STATE` structure.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.15 USBD_CDC_Write()

Description

Writes data to the host. Depending on the Timeout parameter, the function may block until NumBytes have been written or a timeout occurs.

Prototype

```
int USBD_CDC_Write(      USB_CDC_HANDLE   hInst,
                   const void           * pData,
                         unsigned         NumBytes,
                         int              Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pData	Pointer to data that should be sent to the host.
NumBytes	Number of bytes to be written.
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0	Successfully started an asynchronous write transfer or a timeout has occurred and no data was written.
> 0 && < NumBytes	Number of bytes that have been written before a timeout occurred.
= NumBytes	Write transfer successfully completed.
< 0	An error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

The USB stack is able to queue a small number of asynchronous write transfers (Timeout = -1). If a write transfer is still in progress when this function is called and the USB stack can not accept another write transfer request, the functions returns USB_STATUS_EP_BUSY. A synchronous write transfer (Timeout ≥ 0) will always block until the transfer (including all pending transfers) are finished.

In order to synchronize, USBD_CDC_WaitForTX() needs to be called. Another synchronization method would be to periodically call USBD_CDC_GetNumBytesRemToWrite() in order to see how many bytes still need to be written (this method is preferred when a non-blocking solution is necessary).

The write operation can be canceled using USBD_CDC_CancelWrite().

If pData = NULL and NumBytes = 0, a zero-length packet is sent to the host.

In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 136).

The content of the buffer pointed to by pData must not be changed until the transfer has been completed.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.16 USBD_CDC_WriteAsync()

Description

Sends data to the host asynchronously. The function does not wait for the data to be sent. A callback function is called after the transfer has completed successfully, an error occurred or the transfer was canceled.

Prototype

```
void USBD_CDC_WriteAsync(USB_CDC_HANDLE         hInst,
                         USB_ASYNC_IO_CONTEXT * pContext,
                         I8                     Send0PacketIfRequired);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
pContext	Pointer to an I/O context containing parameters and pointer to the callback function.
Send0PacketIfRequired	Specifies that a zero-length packet shall be sent when the last data packet is a multiple of MaxPacketSize.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.17 USBD_CDC_WaitForRX()

Description

This function is to be used in combination with USBD_CDC_ReadOverlapped() and waits for the reading data transfer from the host to complete.

Prototype

```
int USBD_CDC_WaitForRX(USB_CDC_HANDLE hInst,
                       unsigned       Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

0 Transfer completed.
1 Timeout occurred.

Additional information

This function shall be called in order to synchronize task with the read data transfer previously initiated. The function blocks until the number of bytes specified by USBD_CDC_ReadOverlapped() has been read from the host. In case of a timeout, the read transfer is aborted (see *Timeout handling* on page 136).

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.18 USBD_CDC_WaitForTX()

Description

This function is to be used in combination with a non-blocking call to USBD_CDC_Write(). This function waits for the writing data transfer to the host to complete.

Prototype

```
int USBD_CDC_WaitForTX(USB_CDC_HANDLE hInst,
                       unsigned       Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

0 Transfer completed.
1 Timeout occurred.

Additional information

This function shall be called in order to synchronize task with the write data transfer previously initiated. This function blocks until the number of bytes specified by USBD_CDC_Write() has been written to the host. In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 136).

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.19 USBD_CDC_PollForTX()

Description

This function is to be used in combination with a non-blocking call to USBD_CDC_Write(). This function waits for the writing data transfer to the host to complete.

Prototype

```
int USBD_CDC_PollForTX(USB_CDC_HANDLE hInst,
                       unsigned       Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

0 Transfer completed.
1 Timeout occurred.

Additional information

This function shall be called in order to synchronize task with the write data transfer previously initiated. This function blocks until the number of bytes specified by USBD_CDC_Write() has been written to the host. In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example

```
if (USBD_CDC_Write(hInst, &ac[0], 50, -1) < 0) {
  <.. error handling..>
  return;
}
//
// USBD_CDC_Write() will return immediately.
// While waiting for the data to be transferred, we will blink a LED with
 200 ms interval.
// USBD_CDC_PollForTX() will return, if all data were send or 100 ms expired.
//
while ((r = USBD_CDC_PollForTX(hInst, 100)) > 0) {
  ToggleLED();
}
if (r < 0) {
  <.. error handling..>
  return;
}
// Now all data have been send.
```

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.20 USBD_CDC_WaitForTXReady()

Description

Waits (blocking) until the TX queue can accept another data packet. This function is used in combination with a non-blocking call to USBD_CDC_Write(), it waits until a new asynchronous write data transfer will be accepted by the USB stack.

Prototype

```
int USBD_CDC_WaitForTXReady(USB_CDC_HANDLE hInst,
                            int            Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is negative, the function will return immediately.

Return value

= 0 A new asynchronous write data transfer will be accepted.
= 1 The write queue is full, a call to USBD_CDC_Write() would return USB_STATUS_EP_BUSY.
< 0 Error occurred.

Additional information

If Timeout is 0, the function never returns 1.

If Timeout is -1, the function will not wait, but immediately return the current state.

Example

```
if (USBD_CDC_ReadOverlapped(hInst, &ac[0], 50) < 0) {
  <.. error handling..>
  return;
}
//
// USBD_CDC_ReadOverlapped() will return immediately.
// Do something else while data may be transferred.
//
<...>
//
// Now wait until we get all 50 bytes.
// USBD_CDC_WaitForRX() will block until a total of
// 50 bytes are read or a timeout occurs.
//
if (USBD_CDC_WaitForRX(hInst, timeout) != 0) {
  <.. timeout error handling..>
  return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.21 USBD_CDC_PollForRX()

Description

This function is to be used in combination with USBD_CDC_ReadOverlapped() and waits for the reading data transfer from the host to complete.

Prototype

```
int USBD_CDC_PollForRX(USB_CDC_HANDLE hInst,
                       unsigned       Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

0 Transfer completed.
1 Timeout occurred.

Additional information

This function shall be called in order to synchronize task with the read data transfer previously initiated. The function blocks until the number of bytes specified by USBD_CDC_ReadOverlapped() has been read from the host. In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example

```
if (USBD_CDC_ReadOverlapped(hInst, &ac[0], 50) < 0) {
  <.. error handling..>
  return;
}
//
// USBD_CDC_ReadOverlapped() will return immediately.
// While waiting for the data, we will blink a LED with 200 ms interval.
// USBD_CDC_PollForRX() will return, if all data were read or 100 ms expired.
//
while ((r = USBD_CDC_PollForRX(hInst, 100)) > 0) {
  ToggleLED();
}
if (r < 0) {
  <.. error handling..>
  return;
}
// Now we have 50 bytes of data.
// Process 50 bytes of data from ac[] here.
```

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.22 USBD_CDC_WaitForTX()

Description

This function is to be used in combination with a non-blocking call to USBD_CDC_Write(). This function waits for the writing data transfer to the host to complete.

Prototype

```
int USBD_CDC_WaitForTX(USB_CDC_HANDLE hInst,
                       unsigned       Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

0 Transfer completed.
1 Timeout occurred.

Additional information

This function shall be called in order to synchronize task with the write data transfer previously initiated. This function blocks until the number of bytes specified by USBD_CDC_Write() has been written to the host. In case of a timeout, the write transfer is aborted (see *Timeout handling* on page 136).

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.23 USBD_CDC_PollForTX()

Description

This function is to be used in combination with a non-blocking call to USBD_CDC_Write(). This function waits for the writing data transfer to the host to complete.

Prototype

```
int USBD_CDC_PollForTX(USB_CDC_HANDLE hInst,
                       unsigned       Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

0 Transfer completed.
1 Timeout occurred.

Additional information

This function shall be called in order to synchronize task with the write data transfer previously initiated. This function blocks until the number of bytes specified by USBD_CDC_Write() has been written to the host. In case of a timeout, the current transfer is not affected. The function may be called repeatedly until it does not report a timeout any more.

Example

```
if (USBD_CDC_Write(hInst, &ac[0], 50, -1) < 0) {
  <.. error handling..>
  return;
}
//
// USBD_CDC_Write() will return immediately.
// While waiting for the data to be transferred, we will blink a LED with
 200 ms interval.
// USBD_CDC_PollForTX() will return, if all data were sent or 100 ms expired.
//
while ((r = USBD_CDC_PollForTX(hInst, 100)) > 0) {
  ToggleLED();
}
if (r < 0) {
  <.. error handling..>
  return;
}
// Now all data have been sent.
```

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.24 USBD_CDC_WaitForTXReady()

Description

Waits (blocking) until the TX queue can accept another data packet. This function is used in combination with a non-blocking call to `USBD_CDC_Write()`, it waits until a new asynchronous write data transfer will be accepted by the USB stack.

Prototype

```
int USBD_CDC_WaitForTXReady(USB_CDC_HANDLE hInst,
                            int            Timeout);
```

Parameters

Parameter	Description
`hInst`	Handle to a valid CDC instance, returned by `USBD_CDC_Add()`.
`Timeout`	`Timeout` in milliseconds. 0 means infinite. If `Timeout` is negative, the function will return immediately.

Return value

= 0 A new asynchronous write data transfer will be accepted.
= 1 The write queue is full, a call to `USBD_CDC_Write()` would return `USB_STATUS_EP_BUSY`.
< 0 Error occurred.

Additional information

If `Timeout` is 0, the function never returns 1.

If `Timeout` is -1, the function will not wait, but immediately return the current state.

Example

```
// Always keep the write queue full for maximum send speed.
for (;;) {
  pData = GetNextData(&NumBytes);
  // Wait until stack can accept a new write.
  USBD_CDC_WaitForTxReady(hInst, 0);
  // Put write transfer into the write queue.
  if (USBD_CDC_Write(hInst, pData, NumBytes, -1) < 0) {
    <.. error handling..>
  }
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.25 USBD_CDC_WriteSerialState()

Description

Sends the current control line serial state to the host. The current control line serial state can be set using `USBD_CDC_UpdateSerialState()`.

This functions returns immediately. The USB transfer is performed asynchronously.

Prototype

```
void USBD_CDC_WriteSerialState(USB_CDC_HANDLE hInst);
```

Parameters

Parameter	Description
`hInst`	Handle to a valid CDC instance, returned by `USBD_CDC_Add()`.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.26 USBD_CDC_GetNumBytesRemToRead()

Description

This function is to be used in combination with `USBD_CDC_ReadOverlapped()`. It returns the number of bytes which still have to be read during the transaction.

Prototype

```
int USBD_CDC_GetNumBytesRemToRead(USB_CDC_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by `USBD_CDC_Add()`.

Return value

Number of bytes which still have to be read.

Additional information

Note that this function does not return the number of bytes that have been read, but the number of bytes which still have to be read. This function does not block.

Example

```
NumBytesReceived = USBD_CDC_ReadOverlapped(hInst, &ac[0], 50);
if (NumBytesReceived < 0) {
  <.. error handling..>
}
if (NumBytesReceived > 0) {
  // Already had some data in the internal buffer.
  // The first 'NumBytesReceived' bytes may be processed here.
  <...>
} else {
  // Wait until we get all 50 bytes
  while (USBD_CDC_GetNumBytesRemToRead(hInst) > 0) {
    USB_OS_Delay(50);
  }
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.27 USBD_CDC_GetNumBytesRemToWrite()

Description

This function is to be used in combination with a non-blocking call to `USBD_CDC_Write()`. It returns the number of bytes which still have to be written during the transaction.

Prototype

```
int USBD_CDC_GetNumBytesRemToWrite(USB_CDC_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by `USBD_CDC_Add()`.

Return value

Number of bytes which still have to be written.

Additional information

Note that this function does not return the number of bytes that have been written, but the number of bytes which still have to be written. This function does not block.

Example

```
// NumBytesWritten will contain > 0 values
// if we had anything in the write buffer.
NumBytesWritten = USBD_CDC_Write(hInst, &ac[0], TRANSFER_SIZE, -1);
if (NumBytesWritten < 0) {
  <.. error handling..>
}
// NumBytesToWrite shows how many bytes still have to be written.
while (USBD_CDC_GetNumBytesRemToWrite(hInst) > 0) {
  USB_OS_Delay(50);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.1.28 USBD_CDC_GetNumBytesInBuffer()

Description

Returns the number of bytes that are available in the internal BULK-OUT endpoint buffer. This function does not start a read transfer.

Prototype

```
int USBD_CDC_GetNumBytesInBuffer(USB_CDC_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid CDC instance, returned by USBD_CDC_Add().

Return value

Number of bytes which have been stored in the internal buffer.

Additional information

The number of bytes returned by this function can be read using USBD_CDC_Read() without blocking.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.2 Data structures

10.3.2.1 USB_CDC_INIT_DATA

Description

Initialization structure that is needed when adding a CDC interface to emUSB-Device.

Type definition

```
typedef struct {
  U8   EPIn;
  U8   EPOut;
  U8   EPInt;
} USB_CDC_INIT_DATA;
```

Structure members

Member	Description
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host.
EPInt	Interrupt IN endpoint for sending status information.

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.2.2 USB_CDC_LINE_CODING

Description

Structure that contains the new line-coding information sent by the host.

Type definition

```
typedef struct {
  U32   DTERate;
  U8    CharFormat;
  U8    ParityType;
  U8    DataBits;
} USB_CDC_LINE_CODING;
```

Structure members

Member	Description
DTERate	The data transfer rate for the device in bits per second.
CharFormat	Number of stop bits: • 0 - 1 Stop bit • 1 - 1.5 Stop bits • 2 - 2 Stop bits
ParityType	Specifies the parity type: • 0 - None • 1 - Odd • 2 - Even • 3 - Mark • 4 - Space
DataBits	Specifies the bits per byte: (5, 6, 7, 8)

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.2.3 USB_CDC_SERIAL_STATE

Description

Structure that contains the serial state that can be sent to the host.

Type definition

```
typedef struct {
  U8   DCD;
  U8   DSR;
  U8   Break;
  U8   Ring;
  U8   FramingError;
  U8   ParityError;
  U8   OverRunError;
  U8   CTS;
} USB_CDC_SERIAL_STATE;
```

Structure members

Member	Description
DCD	Data Carrier Detect: Tells that the device is connected to the telephone line.
DSR	Data Set Read: Device is ready to receive data.
Break	1 - Break condition signaled.
Ring	Device indicates that it has detected a ring signal on the telephone line.
FramingError	When set to 1, the device indicates a framing error.
ParityError	When set to 1, the device indicates a parity error.
OverRunError	When set to 1, the device indicates an over-run error.
CTS	Clear to Send: Deprecated, not used with USB.

Additional information

All members of the structure may have value 0 (false) or 1 (true).

 © 2010-2026 SEGGER Microcontroller GmbH

10.3.2.4 USB_CDC_CONTROL_LINE_STATE

Description

Structure that contains the new control line state sent by the host.

Type definition

```
typedef struct {
  U8   DTR;
  U8   RTS;
} USB_CDC_CONTROL_LINE_STATE;
```

Structure members

Member	Description
DTR	Data Terminal Ready.
RTS	Request To Send.

Additional information

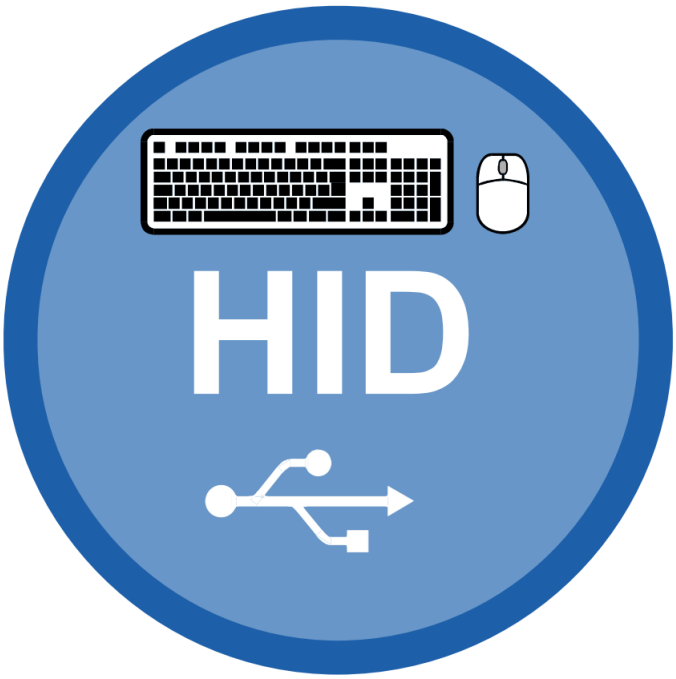
All members of the structure may have value 0 (false) or 1 (true).

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 11

Human Interface Device Class (HID)

This chapter gives a general overview of the HID class and describes how to get the HID component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

11.1 Overview

The Human Interface Device class (HID) is an abstract USB class protocol defined by the USB Implementers Forum. This protocol was defined for the handling of devices which are used by humans to control the operation of computer systems.

An installation of a custom-host USB driver is not necessary, because the USB Human Interface Device class is standardized and every major OS already provides host drivers for it.

11.1.1 Further reading

The following documents define the HID class and have been used to implement and verify the HID component:

- [HID1] Device Class Definition for Human Interface Devices (HID), Firmware Specification—6/27/01 Version 1.11
- [HID2] HID Usage Tables, 1/21/2005 Version 1.12

11.1.2 Categories

Devices which are in the HID class generally fall into one of two categories:

True HIDs and *vendor-specific HIDs*, explained below. One or more examples for both categories are provided.

11.1.2.1 True HIDs

True HID devices are devices which communicate directly with the host operating system. This includes devices which are used by a human to enter data, but do not directly exchange data with an application program running on the host.

Typical examples

- Keyboard
- Mouse and similar pointing devices
- Joystick
- Gamepad
- Bar-code reader
- Front-panel controls - for example, switches and buttons.

11.1.2.2 Vendor specific HIDs

These are HID devices communicating with an application program. The host OS loads the same driver it loads for any “true HID” and will automatically enumerate the device, but it cannot communicate with the device. When analyzing the report descriptor, the host finds that it cannot exchange information with the device; the device uses a protocol which is meaningless to the HID driver of the host. The host will therefore not exchange information with the device. A host recognizes a vendor specific HID by its vendor-defined usage page in the report descriptor: the numerical value of the usage page lies between `0xFF00` and `0xFFFF`.

An application has the chance to communicate with the particular device using API functions offered by the host. This enables an application program to communicate with the device without having to load a driver. HID does not take advantage of the full USB bus bandwidth; bulk communication can be much faster, but requires a driver with older operating systems. Therefore, it can be a good choice to select HID as a device class, especially if ease of use is important and high communication speed is not required.

Typical examples

- Thermometer
- Voltmeter
- Low-speed JTAG emulator

 © 2010-2026 SEGGER Microcontroller GmbH

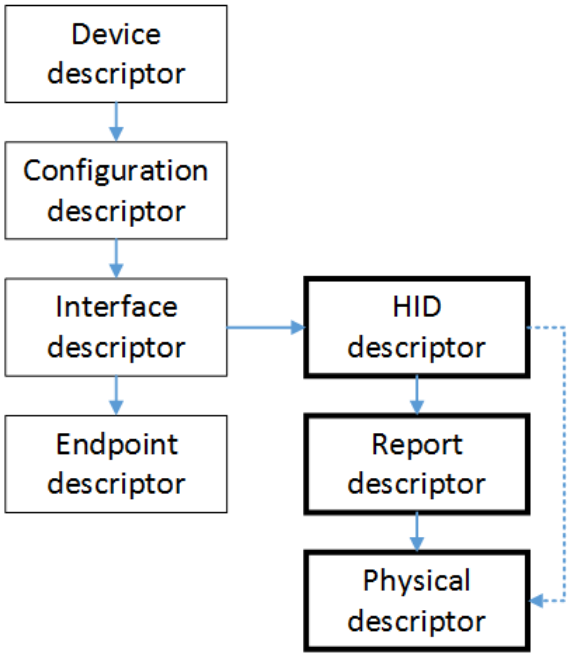
- UPS (Uninterruptible power supply)

 © 2010-2026 SEGGER Microcontroller GmbH

11.2 Background information

11.2.1 HID descriptors

This section presents an overview of the HID class-specific descriptors. The HID descriptors are defined in the *Device Class Definition for Human Interface Devices (HID)* of the USB Implementers Forum. Refer to the USB Implementers Forum website, https://www.usb.org, for detailed information about the USB HID standard.



11.2.1.1 HID descriptor

A HID descriptor contains the report descriptor and optionally the physical descriptors. It specifies the number, type, and size of the report descriptor and the report's physical descriptors.

11.2.1.2 Report descriptor

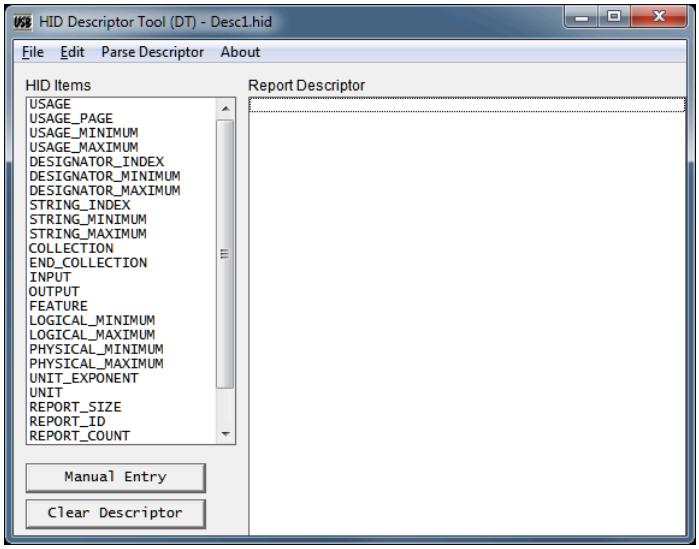
Data between the host and the device is exchanged in so-called "reports". The report descriptor defines the format of a report. In general, HIDs require a report descriptor as defined in the Device *Class Definition for Human Interface Devices (HID)*. The only exception to this are very basic HIDs such as mice or keyboards. This implementation of HID always requires a report descriptor.

When using HID, only transfers matching the report size are allowed. For example, if a report is defined to be 64 bytes large in either direction, only transfers of 64 bytes are allowed. If the application needs to transfer less data, the packet must be padded by the application to match the report size. The report descriptor can define multiple reports of different sizes. In this case, the first byte of the transfer must contain the report ID.

The USB Implementers Forum provides an application which helps to build and modify HID report descriptors. The HID Descriptor Tool can be downloaded from:

https://www.usb.org/hid

 © 2010-2026 SEGGER Microcontroller GmbH



11.2.1.3 Physical descriptor

Physical descriptor sets are optional descriptors which provide information about the part or parts of the human body used to activate the controls on a device. Physical descriptors are currently not supported.

 © 2010-2026 SEGGER Microcontroller GmbH

11.3 Configuration

11.3.1 Initial configuration

To get emUSB-Device up and running as well as to do an initial test, the configuration as it is delivered should not be modified. The configuration must only be modified if emUSB-Device should be used in your final product. Refer to the section *emUSB-Device Configuration* on page 53 for detailed information about the functions which must be adapted before you can release a final product version.

11.3.2 Final configuration

Generating a report descriptor

This step is only required if your product is a vendor-specific human interface device. The report descriptor provided in the example application can typically be used without any modification. The vendor-defined usage page should be adapted in a final product. Vendor-defined usage pages can be in the range from `0xFF00` to `0xFFFF`. The low byte can be selected by the application programmer. It needs to be identical on both target and host and should be unique (as unique as an 8-bit value can be). The examples use the value `0x12`; this value is defined at the top of the application program with the macro `USB_HID_DEFAULT_VENDOR_PAGE`.

 © 2010-2026 SEGGER Microcontroller GmbH

11.4 Example application

Example applications are supplied. These can be used for testing the correct installation and proper function of the device running emUSB-Device.

The following start application files are provided:

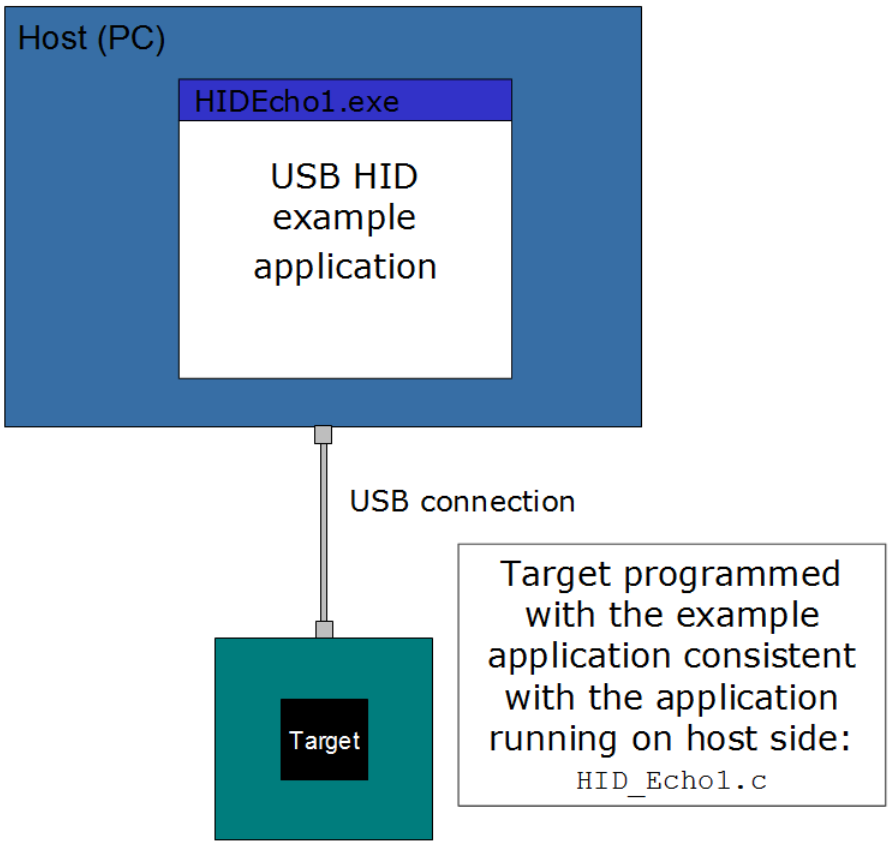
File	Description
USB_HID_Mouse.c	Simple mouse example. (“True HID” example)
USB_HID_Echo1.c	Modified echo server. (“vendor-specific” example)

11.4.1 USB_HID_Mouse.c

USB_HID_Mouse.c is a typical example of a “true HID” implementation. The host identifies the device which is programmed with this example as a mouse. After the device is enumerated, it moves the mouse cursor in an endless loop to the left and after a short delay back to the right.

11.4.2 USB_HID_Echo1.c

USB_HID_Echo1.c is a typical example for a “vendor-specific HID” implementation. The HID start application (USB_HID_Echo1.c located in the Application subfolder) is a modified echo server; the application receives data byte by byte, increments every single byte and sends them back to the host.



To use this application, include the source code file USB_HID_Echo1.c into your project and compile and download it into your target. Run HIDEcho1.exe after the target is connected to the host and the enumeration process has been completed. The PC application is supplied as executable in the Windows\USB\HID\SampleApp\Exe directory. The source code of the

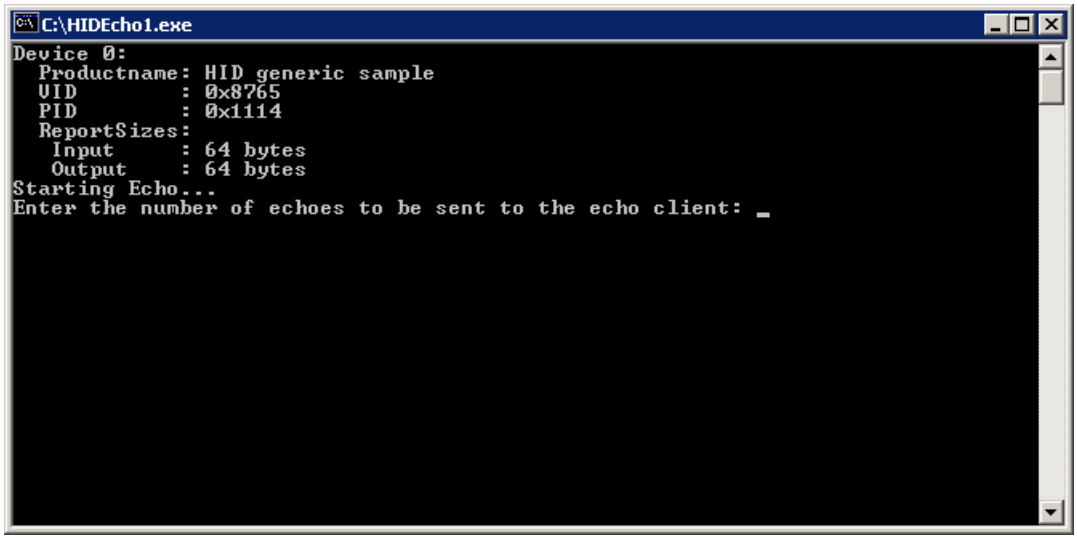
 © 2010-2026 SEGGER Microcontroller GmbH

PC example is also supplied. Refer to section Compiling the PC example application for more information about the PC example project.

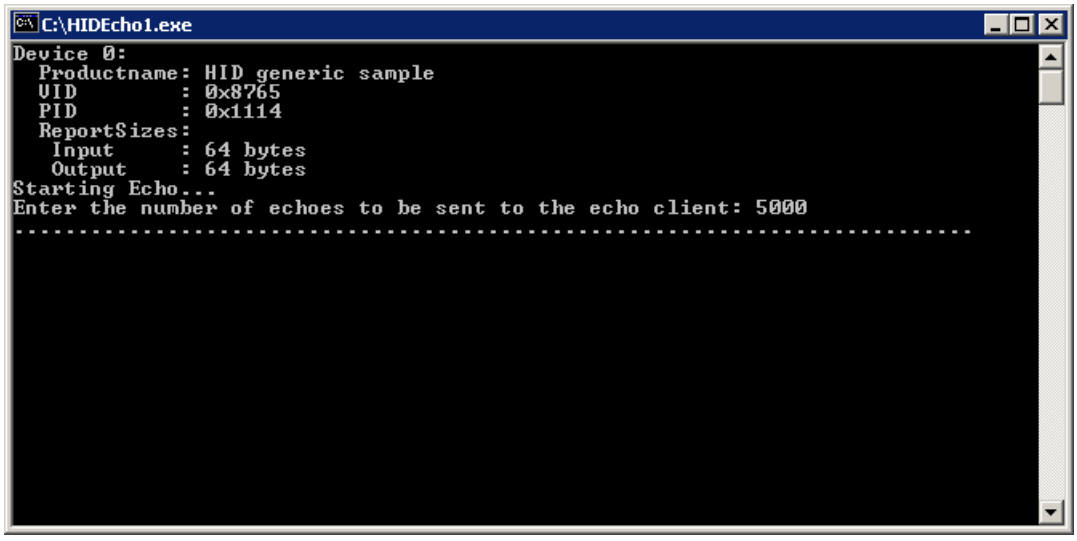
11.4.2.1 Running the example

1. Add `USB_HID_Echo1.c` to your project and build and download the application into the target.
2. Connect your target to the host via USB while the example application is running. Windows will detect the new HID device.
3. If a connection can be established, it exchanges data with the target, testing the USB connection. If the host example application can communicate with the emUSB-Device, the example application outputs the product name, Vendor and Product ID and the report size which will be used to communicate with the target. The target will be in interactive mode.

Example output of `USB_HID_Echo1.exe`:



4. Enter the number of reports that should be transmitted when the device is connected. Every dot in the terminal window indicates a transmission.



11.4.2.2 Compiling the PC example application

Under Windows, you can build the sample by using the provided Visual Studio 2010 project. The source code of the example application is located in the subfolder `Windows\USB\HID\SampleApp`. Open the file `USBHID_Start.sln` and, to compile the source, choose **Build | Build SampleApp.exe** (Shortcut: F7). To run the executable, choose **Build | Execute SampleApp.exe** (Shortcut: CTRL-F5).

 © 2010-2026 SEGGER Microcontroller GmbH

> **Note**
>
> The Microsoft Windows Driver Development Kit (DDK) is required to compile the HID host example application. Refer to https://docs.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk for more information.

Under Linux, simply generate the executable by invoking make in the `Windows/USB/HID/SampleApp` folder in a shell

```
cd Windows/USB/HID/SampleApp
make
```

 © 2010-2026 SEGGER Microcontroller GmbH

11.5 Target API

This section describes the functions that can be used on the target system.

General information

To communicate with the host, the example application project includes USB-specific header and source files. These files contain API functions to communicate with the USB host.

Purpose of the USB Device API functions

To have an easy start-up when writing an application on the device side, these API functions have a simple interface and handle all operations that need to be done to communicate with the host. Therefore, all operations that need to write to or read from the emUSB-Device are handled internally by the provided API functions.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.1 Target interface function list

Function	Description
API functions	
USBD_HID_AddEx()	Adds HID class device to the USB interface.
USBD_HID_Add()	Adds HID class device to the USB interface.
USBD_HID_GetNumBytesInBuffer()	Returns the number of bytes available in the internal read buffer.
USBD_HID_GetNumBytesRemToRead()	Checks how many bytes still have to be read.
USBD_HID_GetNumBytesRemToWrite()	Checks how many bytes still have to be written.
USBD_HID_Read()	Reads data from the host with a given timeout.
USBD_HID_ReadOverlapped()	Reads data from the host asynchronously.
USBD_HID_Receive()	Reads data from the host.
USBD_HID_ReceivePoll()	Reads data from the host.
USBD_HID_WaitForRX()	This function is to be used in combination with USBD_HID_ReadOverlapped().
USBD_HID_WaitForTX()	This function is to be used in combination with a non-blocking call to USBD_HID_Write().
USBD_HID_Write()	Writes data to the host.
USBD_HID_SetOnGetReportRequest()	Allows to set a callback for the GET_REPORT command.
USBD_HID_SetOnSetReportRequest()	Allows to set a callback for the SET_REPORT control command.
USBD_HID_ReadReport()	Reads report data that was sent from the host via the control EP.
Data structures	
USB_HID_INIT_DATA_EX	Initialization structure that is needed when adding a HID interface to emUSB-Device.
USB_HID_INIT_DATA	Initialization structure that is needed when adding a HID interface to emUSB-Device.
Type definitions	
USB_HID_ON_GETREPORT_REQUEST_FUNC	Callback function which is set via USBD_HID_SetOnGetReportRequest().
USB_HID_ON_SETREPORT_REQUEST_FUNC	Callback function which is set via USBD_HID_SetOnSetReportRequest().

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2 HID Target API functions

11.5.2.1 USBD_HID_AddEx()

Description

Adds HID class device to the USB interface.

Prototype

```
USB_HID_HANDLE USBD_HID_AddEx(const USB_HID_INIT_DATA_EX * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a USB_HID_INIT_DATA_EX structure. For detailed information refer to USB_HID_INIT_DATA_EX.

Return value

USB_HID_HANDLE: Handle to the HID instance (can be zero).

Additional information

After the initialization of general emUSB-Device, this is the first function that needs to be called when the USB-HID interface is used with emUSB-Device.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.2 USBD_HID_Add()

Description

Adds HID class device to the USB interface.

Prototype

```
USB_HID_HANDLE USBD_HID_Add(const USB_HID_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a USB_HID_INIT_DATA structure. For detailed information refer to USB_HID_INIT_DATA.

Return value

USB_HID_HANDLE: Handle to the HID instance (can be zero).

Additional information

After the initialization of general emUSB-Device, this is the first function that needs to be called when the USB-HID interface is used with emUSB-Device.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.3 USBD_HID_GetNumBytesInBuffer()

Description

Returns the number of bytes available in the internal read buffer.

Prototype

```
unsigned USBD_HID_GetNumBytesInBuffer(USB_HID_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to an HID handle which is returned by USBD_HID_Add().

Return value

≥ 0 Number of bytes in the internal read buffer.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.4 USBD_HID_GetNumBytesRemToRead()

Description

Checks how many bytes still have to be read.

Prototype

```
unsigned USBD_HID_GetNumBytesRemToRead(USB_HID_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to an HID instance.

Return value

≥ 0 Number of bytes which have not yet been read.

Additional information

This function is to be used in combination with `USBD_HID_ReadOverlapped()`. After starting the read operation this function can be used to periodically check how many bytes still have to be read. Alternatively the blocking function `USBD_HID_WaitForRX()` can be used.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.5 USBD_HID_GetNumBytesRemToWrite()

Description

Checks how many bytes still have to be written.

Prototype

```
unsigned USBD_HID_GetNumBytesRemToWrite(USB_HID_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to an HID instance.

Return value

≥ 0 Number of bytes which have not yet been written.

Additional information

This function is to be used in combination with a non-blocking call to `USBD_HID_Write()`. After starting the write operation this function can be used to periodically check how many bytes still have to be written. Alternatively the blocking function `USBD_HID_WaitForTX()` can be used.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.6 USBD_HID_Read()

Description

Reads data from the host with a given timeout.

Prototype

```
int USBD_HID_Read(USB_HID_HANDLE   hInst,
                  void           * pData,
                  unsigned         NumBytes,
                  unsigned         Timeout);
```

Parameters

Parameter	Description
hInst	Handle to an HID instance.
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
Timeout	Timeout given in milliseconds. A zero value results in an infinite timeout.

Return value

= NumBytes	Requested data was successfully read within the given timeout.
≥ 0, < NumBytes	Timeout has occurred. Number of bytes that have been read within the given timeout.
< 0	Returns a USB_STATUS_ERROR.

Additional information

This function blocks until the timeout has been reached, it has received NumBytes or until the device is disconnected from the host. This function blocks a task until all data has been read or a timeout occurs. In case of a reset or a disconnect USB_STATUS_ERROR is returned.

The host will always send transmissions which match the report size. In most cases it makes sense to set NumBytes to the report size.

When using multiple reports the first byte will contain the report ID.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.7 USBD_HID_ReadOverlapped()

Description

Reads data from the host asynchronously.

Prototype

```
int USBD_HID_ReadOverlapped(USB_HID_HANDLE   hInst,
                            void           * pData,
                            unsigned         NumBytes);
```

Parameters

Parameter	Description
hInst	Handle to a HID instance.
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Return value

> 0	Number of bytes that have been read from the internal buffer (success).
= 0	No data was found in the internal buffer (success).
< 0	Error.

Additional information

This function will not block the calling task. The read transfer will be initiated and the function returns immediately. In order to synchronize, USBD_HID_WaitForRX() needs to be called. Alternatively the function USBD_HID_GetNumBytesRemToRead() can be called periodically to check whether all bytes have been written or not. The buffer pointed to by pData must be valid until the read operation is terminated.

The host will always send transmissions which match the report size. In most cases it makes sense to set NumBytes to the report size.

When using multiple reports the first byte will contain the report ID.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.8 USBD_HID_Receive()

Description

Reads data from the host. The function blocks until any data has been received or a timeout occurs (if `Timeout` ≥ 0). In contrast to `USBD_HID_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received. In case of a timeout, the read transfer is aborted (see *`Timeout` handling* on page 136).

Prototype

```
int USBD_HID_Receive(USB_HID_HANDLE   hInst,
                     void           * pData,
                     unsigned         NumBytes,
                     int              Timeout);
```

Parameters

Parameter	Description
`hInst`	Handle to an HID instance.
`pData`	Pointer to a buffer where the received data will be stored.
`NumBytes`	Maximum number of bytes to read.
`Timeout`	`Timeout` in milliseconds. 0 means infinite. If `Timeout` is -1, the function never blocks and only reads data from the internal endpoint buffer.

Return value

> 0	Number of bytes that have been read.
= 0	A timeout occurred (if `Timeout` > 0), zero packet received (not every controller supports this!), no data in buffer (if `Timeout` < 0) or the target was disconnected during the function call and no data was read so far.
< 0	Error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USBD_HID_Receive()` will return as much data as is currently available -- up to the size of the buffer specified. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return `USB_STATUS_ERROR`.

If a read transfer was pending while the function is called, it returns `USB_STATUS_EP_BUSY`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBD_AddEP()`. This data can be retrieved by a later call to `USBD_HID_Receive()` / `USBD_HID_Read()`. See also `USBD_HID_GetNumBytesInBuffer()`.

A call of `USBD_HID_Receive(Inst, NULL, 0, -1)` can be used to trigger an asynchronous read that stores the data into the internal buffer.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.9 USBD_HID_ReceivePoll()

Description

Reads data from the host. The function blocks until any data has been received or a timeout occurs (if `Timeout` ≥ 0). In contrast to `USBD_BULK_Read()` this function does not wait for all of `NumBytes` to be received, but returns after the first packet has been received. In contrast to `USBD_BULK_Receive()` this function will continue the read transfer asynchronously in case of a timeout.

Prototype

```
int USBD_HID_ReceivePoll(USB_HID_HANDLE   hInst,
                         void           * pData,
                         unsigned         NumBytes,
                         unsigned         Timeout);
```

Parameters

Parameter	Description
`hInst`	Handle to an HID instance.
`pData`	Pointer to a buffer where the received data will be stored.
`NumBytes`	Maximum number of bytes to read.
`Timeout`	`Timeout` in milliseconds. 0 means infinite.

Return value

> 0	Number of bytes that have been read.
= 0	A timeout occurred (if `Timeout` > 0), zero packet received (not every controller supports this!) or the target was disconnected during the function call and no data was read so far.
< 0	Error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling `USBD_HID_ReceivePoll()` will return as much data as is currently available -- up to the size of the buffer specified. This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return `USB_STATUS_ERROR`.

If a read transfer was pending while the function is called, it returns `USB_STATUS_EP_BUSY`.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via `USBD_AddEP()`. This data can be retrieved by a later call to `USBD_HID_Receive()` / `USBD_HID_Read()`. See also `USBD_HID_GetNumBytesInBuffer()`.

If a timeout occurs, the read transfer is not affected. Data sent from the host after the timeout is stored into the internal buffer of the endpoint and can be read by later calls to `USBD_HID_ReceivePoll()`.

If `Timeout` = 0, the function behaves like `USBD_HID_Receive()`.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.10 USBD_HID_WaitForRX()

Description

This function is to be used in combination with `USBD_HID_ReadOverlapped()`. After the read function has been called this function can be used to synchronize. It will block until the transfer is completed.

Prototype

```
int USBD_HID_WaitForRX(USB_HID_HANDLE hInst,
                       unsigned       Timeout);
```

Parameters

Parameter	Description
`hInst`	Handle to a HID instance.
`Timeout`	`Timeout` given in milliseconds. A zero value results in an infinite timeout.

Return value

= 0 Transfer completed.
= 1 `Timeout` occurred.
< 0 Error occurred.

Additional information

In case of a timeout, a current transfer is canceled.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.11 USBD_HID_WaitForTX()

Description

This function is to be used in combination with a non-blocking call to `USBD_HID_Write()`. After the write function has been called this function can be used to synchronize. It will block until the transfer is completed.

Prototype

```
int USBD_HID_WaitForTX(USB_HID_HANDLE hInst,
                       unsigned       Timeout);
```

Parameters

Parameter	Description
`hInst`	Handle to a HID instance.
`Timeout`	`Timeout` given in milliseconds. A zero value results in an infinite timeout.

Return value

= 0 Transfer completed.
= 1 `Timeout` occurred.
< 0 An error occurred.

Additional information

In case of a timeout, a current transfer is canceled.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.12 USBD_HID_Write()

Description

Writes data to the host. Depending on the `Timeout` parameter, the function may block until `NumBytes` have been written or a timeout occurs.

Prototype

```
int USBD_HID_Write(      USB_HID_HANDLE   hInst,
                   const void           * pData,
                         unsigned         NumBytes,
                         int              Timeout);
```

Parameters

Parameter	Description
`hInst`	Handle to an HID instance.
`pData`	Pointer to data that should be sent to the host.
`NumBytes`	Number of bytes to write. Should match the report size.
`Timeout`	`Timeout` in milliseconds. 0 means infinite. If `Timeout` is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0	Successfully started an asynchronous write transfer or a timeout has occurred and no data was written.
> 0 && < `NumBytes`	Number of bytes that have been written before a timeout occurred.
= `NumBytes`	Write transfer successfully completed.
< 0	Error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

The USB stack is able to queue a small number of asynchronous write transfers (`Timeout` = -1). If a write transfer is still in progress when this function is called and the USB stack can not accept another write transfer request, the functions returns `USB_STATUS_EP_BUSY`.

In order to synchronize, `USBD_HID_WaitForTX()` needs to be called. Another synchronization method would be to periodically call `USBD_HID_GetNumBytesRemToWrite()` in order to see how many bytes still need to be written (this method is preferred when a non-blocking solution is necessary).

The content of the buffer pointed to by `pData` must not be changed until the transfer has been completed.

A transfer which does not match the report size will not be accepted by the host.

When using multiple reports the first byte must contain the report ID.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.13 USBD_HID_SetOnGetReportRequest()

Description

Allows to set a callback for the GET_REPORT command. The GET_REPORT command is sent from the host to the device.

Prototype

```
void USBD_HID_SetOnGetReportRequest
                        (USB_HID_HANDLE                      hInst,
                         USB_HID_ON_GETREPORT_REQUEST_FUNC * pfOnGetReportRequest);
```

Parameters

Parameter	Description
hInst	Handle to an HID instance.
pfOnGetReportRequest	Pointer to a function of type USB_HID_ON_GETREPORT_REQUEST_FUNC.

Additional information

See the description of USB_HID_ON_GETREPORT_REQUEST_FUNC for more details.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.14 USBD_HID_SetOnSetReportRequest()

Description

Allows to set a callback for the SET_REPORT control command. The SET_REPORT command is sent from the host to the device.

Prototype

```
void USBD_HID_SetOnSetReportRequest
                        (USB_HID_HANDLE                      hInst,
                         USB_HID_ON_SETREPORT_REQUEST_FUNC * pfOnSetReportRequest);
```

Parameters

Parameter	Description
hInst	Handle to an HID instance.
pfOnSetReportRequest	Pointer to a function of type USB_HID_ON_SETREPORT_REQUEST_FUNC .

Additional information

See the description of USB_HID_ON_SETREPORT_REQUEST_FUNC for more details.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.2.15 USBD_HID_ReadReport()

Description

Reads report data that was sent from the host via the control EP. This function returns immediately and will not wait for a report send from the host. Can be used in combination with a callback function installed with `USBD_HID_SetOnSetReportRequest()`.

Prototype

```
int USBD_HID_ReadReport(USB_HID_HANDLE   hInst,
                        void           * pData,
                        unsigned         NumBytes);
```

Parameters

Parameter	Description
`hInst`	Handle to a HID instance.
`pData`	Pointer to a buffer where the received data will be stored.
`NumBytes`	Number of bytes to read.

Return value

> 0	Number of bytes that have been read.
= 0	No report was sent from the host.
< 0	Error.

Additional information

The host will always send transmissions which match the report size. In most cases it makes sense to set `NumBytes` to the report size.

When using multiple reports the first byte will contain the report ID.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.3 Data structures

11.5.3.1 USB_HID_INIT_DATA_EX

Description

Initialization structure that is needed when adding a HID interface to emUSB-Device.

Type definition

```
typedef struct {
  U16           Flags;
  U8            EPIn;
  U8            EPOut;
  const U8    * pReport;
  U16           NumBytesReport;
  U16           BuffSize;
  U8          * pBuff;
  const char  * pInterfaceName;
} USB_HID_INIT_DATA_EX;
```

Structure members

Member	Description
Flags	Reserved, must be set to 0.
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
pReport	Pointer to a report descriptor.
NumBytesReport	Size of the HID report descriptor in bytes.
BuffSize	Size of the buffer pointed to by pBuff. Must be at least the size of the output report.
pBuff	Pointer to a buffer for receiving reports from the host via endpoint 0 (Set_Report request).
pInterfaceName	Name of the interface. May be NULL.

Additional information

To be able to receive input reports from the host either an endpoint must be allocated (EPOut) or a buffer must be provided (BufferSize, pBuff). If both EPOut = 0 and BufferSize = 0, then USBD_HID_Read() will not work and all requests from the host will be stalled by the USB stack. To receive Set Feature Report control commands the buffer is required.

pReport points to a report descriptor. A report descriptor is a structure which is used to transmit HID control data to and from a human interface device. A report descriptor defines the format of a report and is composed of report items that define one or more top-level collections. Each collection defines one or more HID reports. Refer to Universal Serial Bus Specification, 1.0 Version and the latest version of the HID Usage Tables guide for detailed information about HID input, output and feature reports. The USB Implementers Forum provide an application that helps to build and modify HID report descriptors. The HID Descriptor Tool can be downloaded from: www.usb.org/developers/hidpage/. The report descriptor used in the supplied example application HID_Echo1.c should match to the requirements of most "vendor specific HID" applications. The report size is defined to 64 bytes. As mentioned before, interrupt endpoints are limited to at most one packet of at most 64 bytes per frame (on full speed devices).

Example 1 (configure to receive reports via separate endpoint)

```
static void _AddHID(void) {
 USB_HID_INIT_DATA_EX InitData;
 U8 Interval = 10;
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
 static U8 acBuffer[64];

 memset(&InitData, 0, sizeof(InitData));
 InitData.EPIn = USB_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_INT, Interval, NULL, 0);
 InitData.EPOut = USB_AddEP(USB_DIR_OUT, USB_TRANSFER_TYPE_INT, Interval,
                            &acBuffer[0], sizeof(acBuffer));
 InitData.pReport = _aHIDReport;
 InitData.NumBytesReport = sizeof(_aHIDReport);
 InitData.pInterfaceName = "HID interface";
 USBD_HID_AddEx(&InitData);
}
```

Example 2 (configure to receive reports via endpoint 0)

```
static void _AddHID(void) {
 USB_HID_INIT_DATA_EX InitData;
 U8 Interval = 10;
 static U8 acBuffer[64];

 memset(&InitData, 0, sizeof(InitData));
 InitData.EPIn = USB_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_INT, Interval, NULL, 0);
 InitData.pBuff = &acBuffer[0];
 InitData.BufferSize = sizeof(acBuffer);
 InitData.pReport = _aHIDReport;
 InitData.NumBytesReport = sizeof(_aHIDReport);
 InitData.pInterfaceName = "HID interface";
 USBD_HID_AddEx(&InitData);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.3.2 USB_HID_INIT_DATA

Description

Initialization structure that is needed when adding a HID interface to emUSB-Device.

Type definition

```
typedef struct {
  U8          EPIn;
  U8          EPOut;
  const U8  * pReport;
  U16         NumBytesReport;
  U16         BuffSize;
  U8        * pBuff;
} USB_HID_INIT_DATA;
```

Structure members

Member	Description
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
pReport	Pointer to a report descriptor.
NumBytesReport	Size of the HID report descriptor in bytes.
BuffSize	Size of the buffer pointed to by pBuff. Must be at least the size of the output report.
pBuff	Pointer to a buffer for receiving reports from the host via endpoint 0 (Set_Report request).

Additional information

To be able to receive input reports from the host either an endpoint must be allocated (EPOut) or a buffer must be provided (BufferSize, pBuff). If both EPOut = 0 and BufferSize = 0, then USBD_HID_Read() will not work and all requests from the host will be stalled by the USB stack. To receive Set Feature Report control commands the buffer is required.

pReport points to a report descriptor. A report descriptor is a structure which is used to transmit HID control data to and from a human interface device. A report descriptor defines the format of a report and is composed of report items that define one or more top-level collections. Each collection defines one or more HID reports. Refer to Universal Serial Bus Specification, 1.0 Version and the latest version of the HID Usage Tables guide for detailed information about HID input, output and feature reports. The USB Implementers Forum provide an application that helps to build and modify HID report descriptors. The HID Descriptor Tool can be downloaded from: www.usb.org/developers/hidpage/. The report descriptor used in the supplied example application HID_Echo1.c should match to the requirements of most “vendor specific HID” applications.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.4 Type definitions

11.5.4.1 USB_HID_ON_GETREPORT_REQUEST_FUNC

Description

Callback function which is set via `USBD_HID_SetOnGetReportRequest()`.

Type definition

```
typedef int USB_HID_ON_GETREPORT_REQUEST_FUNC(      USB_HID_REPORT_TYPE ReportType,
                                                    unsigned            ReportId,
                                              const U8                * * pData,
                                                    U32                 * pNumBytes);
```

Parameters

Parameter	Description
`ReportType`	HID report type, possible values are: • `USB_HID_REPORT_TYPE_INPUT` • `USB_HID_REPORT_TYPE_OUTPUT` • `USB_HID_REPORT_TYPE_FEATURE`
`ReportId`	The ID of the report for which the `GET_REPORT` request has been sent.
`pData`	in Pointer to a pointer to the data to send via `GET_REPORT` request.
`pNumBytes`	in Number of bytes requested. Out: Number of bytes that shall be sent.

Return value

= 0	No data available. The stack will send a zero length packet as a response.
= 1	Data is available. The stack will send data to the host.
< 0	Data is handled by user application. `USBD_WriteEP0FromISR()` needs to be called from user context.

 © 2010-2026 SEGGER Microcontroller GmbH

11.5.4.2 USB_HID_ON_SETREPORT_REQUEST_FUNC

Description

Callback function which is set via USBD_HID_SetOnSetReportRequest(). The function is called after a SET_REPORT command was sent from the host via the control endpoint. The report should be read using USBD_HID_ReadReport().

Type definition

```
typedef void USB_HID_ON_SETREPORT_REQUEST_FUNC(USB_HID_REPORT_TYPE ReportType,
                                               unsigned            ReportId,
                                               U32                 NumBytes);
```

Parameters

Parameter	Description
ReportType	HID report type, possible values are: • USB_HID_REPORT_TYPE_INPUT • USB_HID_REPORT_TYPE_OUTPUT • USB_HID_REPORT_TYPE_FEATURE
ReportId	The ID of the report for which the SET_REPORT request has been sent.

Additional information

In case no EP Out was used with the HID interface, and a USBD_HID_Read() or USBD_HID_ReadOverlapped() is currently executed, then this function is not called and the read function is serviced instead.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6 Host API

This chapter describes the functions that can be used on the host side (Windows, Linux, macOS). These functions are only required if the emUSB-Device-HID component is used to design a vendor-specific HID.

General information

To communicate with the target USB-HID stack, the example application project includes USB-HID-specific source and header files (USBHID.c, USBHID.h). These files contain API functions to communicate with the USB-HID target through the host HID driver.

Purpose of the USB Host API functions

To have an easy start-up when writing an application on the host side, these API functions have simple interfaces and handle all operations that need to be done to communicate with the target USB-HID stack.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.1 Host API function list

Function	Description
API functions	
USBHID_Close()	Close the connection to an open device.
USBHID_Open()	Opens a handle to a device.
USBHID_Init()	Sets the specific vendor page, initializes the USB HID User API and retrieves the information of the HID device.
USBHID_Exit()	Closes the connection to all open devices and de-initializes the HID module.
USBHID_Read()	Reads an input report from device via the interrupt endpoint.
USBHID_Write()	Writes an output report to device.
USBHID_GetNumAvailableDevices()	Returns the number of available devices.
USBHID_GetProductName()	Stores the name of the device into pBuffer.
USBHID_GetInputReportSize()	Returns the input report size of the device.
USBHID_GetOutputReportSize()	Returns the output report size of the device.
USBHID_GetProductId()	Returns the USB product ID of the device.
USBHID_GetVendorId()	Returns the USB vendor ID of the device.
USBHID_RefreshList()	Refreshes the connection info list.
USBHID_SetVendorPage()	Sets the vendor page so that all HID devices with the specified page will be found.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2 HID Host API functions

11.6.2.1 USBHID_Close()

Description

Close the connection to an open device.

Prototype

```
void USBHID_Close(unsigned Id);
```

Parameters

Parameter	Description
`Id`	Index of the HID device. This is the bit number of the mask returned by `USBHID_GetNumAvailableDevices()`.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.2 USBHID_Open()

Description

Opens a handle to a device.

Prototype

```
int USBHID_Open(unsigned Id);
```

Parameters

Parameter	Description
Id	Index of the HID device. This is the bit number of the mask returned by USBHID_GetNumAvailableDevices().

Return value

0	O.K. Opening was successful or already opened.
1	Error. Handle to the device could not opened.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.3 USBHID_Init()

Description

Sets the specific vendor page, initializes the USB HID User API and retrieves the information of the HID device.

Prototype

```
void USBHID_Init(uint8_t VendorPage);
```

Parameters

Parameter	Description
VendorPage	This parameter specifies the lower 8 bits of the vendor-specific usage page number. It must be identical on both device and host.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.4 USBHID_Exit()

Description

Closes the connection to all open devices and de-initializes the HID module.

Prototype

```
void USBHID_Exit(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.5 USBHID_Read()

Description

Reads an input report from device via the interrupt endpoint.

Prototype

```
int USBHID_Read(unsigned   Id,
                void     * pBuffer,
                unsigned   NumBytes);
```

Return value

On Error: -1, No valid device Id used or the report size does not match with device. On success: Number of bytes that have be written.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.6 USBHID_Write()

Description

Writes an output report to device.

Prototype

```
int USBHID_Write(      unsigned   Id,
                 const void     * pBuffer,
                       unsigned   NumBytes);
```

Return value

On Error: -1, No valid device Id used or the report size does not match with device. On success: Number of bytes that have be written.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.7 USBHID_GetNumAvailableDevices()

Description

Returns the number of available devices.

Prototype

```
unsigned USBHID_GetNumAvailableDevices(uint32_t * pMask);
```

Parameters

Parameter	Description
pMask	Pointer to unsigned integer value which is used to store the bit mask of available devices. This parameter may be NULL.

Return value

Number of available devices.

Additional information

pMask will be filled by this routine. It shall be interpreted as a bit mask where a bit set means this device is available. For example, device 0 and device 2 are available, if pMask has the value 0x00000005.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.8 USBHID_GetProductName()

Description

Stores the name of the device into `pBuffer`.

Prototype

```
int USBHID_GetProductName(unsigned   Id,
                          char     * pBuffer,
                          unsigned   NumBytes);
```

Parameters

Parameter	Description
`Id`	Index of the HID device. This is the bit number of the mask returned by `USBHID_GetNumAvailableDevices()`.
`pBuffer`	Pointer to a buffer for the product name.
`NumBytes`	Size of the buffer in bytes.

Return value

0 An error occurred.
1 Success.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.9 USBHID_GetInputReportSize()

Description

Returns the input report size of the device.

Prototype

```
int USBHID_GetInputReportSize(unsigned Id);
```

Parameters

Parameter	Description
Id	Index of the HID device. This is the bit number of the mask returned by USBHID_GetNumAvailableDevices().

Return value

= 0 An error occurred.
≠ 0 Size of the report in bytes.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.10 USBHID_GetOutputReportSize()

Description

Returns the output report size of the device.

Prototype

```
int USBHID_GetOutputReportSize(unsigned Id);
```

Parameters

Parameter	Description
Id	Index of the HID device. This is the bit number of the mask returned by USBHID_GetNumAvailableDevices().

Return value

= 0 An error occurred.
≠ 0 Size of the report in bytes.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.11 USBHID_GetProductId()

Description

Returns the USB product ID of the device.

Prototype

```
uint16_t USBHID_GetProductId(unsigned Id);
```

Parameters

Parameter	Description
Id	Index of the HID device. This is the bit number of the mask returned by USBHID_GetNumAvailableDevices().

Return value

= 0 An error occurred.
≠ 0 Product ID.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.12 USBHID_GetVendorId()

Description

Returns the USB vendor ID of the device.

Prototype

```
uint16_t USBHID_GetVendorId(unsigned Id);
```

Parameters

Parameter	Description
Id	Index of the HID device. This is the bit number of the mask returned by USBHID_GetNumAvailableDevices().

Return value

= 0 An error occurred.
≠ 0 Vendor ID.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.13 USBHID_RefreshList()

Description

Refreshes the connection info list.

Prototype

```
void USBHID_RefreshList(void);
```

Additional information

Note that any open handles will be closed while refreshing the connection list.

 © 2010-2026 SEGGER Microcontroller GmbH

11.6.2.14 USBHID_SetVendorPage()

Description

Sets the vendor page so that all HID devices with the specified page will be found.

Prototype

```
void USBHID_SetVendorPage(uint8_t Page);
```

Parameters

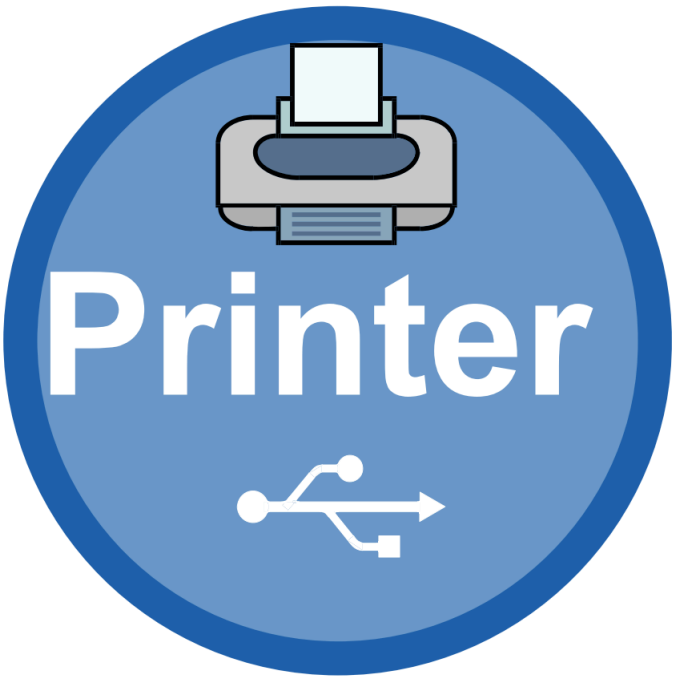
Parameter	Description
Page	This parameter specifies the lower 8 bits of the vendor-specific usage page number. It must be identical on both device and host.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 12

Printer Class

This chapter describes how to get emUSB-Device up and running as a printer device.



 © 2010-2026 SEGGER Microcontroller GmbH

12.1 Overview

The Printer Class is an abstract USB class protocol defined by the USB Implementers Forum. This protocol delivers the existing printing command-sets to a printer over USB.

12.1.1 Configuration

The configuration section will later on be modified to match the real application. For the purpose of getting emUSB-Device up and running as well as doing an initial test, the configuration as delivered should not be modified.

 © 2010-2026 SEGGER Microcontroller GmbH

12.2 The example application

The start application (in the `Application` subfolder) is a simple data sink, which can be used to test emUSB-Device. The application receives data bytes from the host which it displays in the terminal I/O window of the debugger.

Part of the source code of `USB_Printer.c`:

```
<...>
/*********************************************************************
*
*       _GetDeviceIdString
*
*/
static const char * _GetDeviceIdString(void) {
  const char * s = "CLASS:PRINTER;MODEL:HP LaserJet 6MP;"
                   "MANUFACTURER:Hewlett-Packard;"
                   "DESCRIPTION:Hewlett-Packard LaserJet 6MP Printer;"
                   "COMMAND SET:PJL,MLC,PCLXL,PCL,POSTSCRIPT;";
  return s;
}
/*********************************************************************
*
*       _GetHasNoError
*
*/
static U8 _GetHasNoError(void) {
  return 1;
}
/*********************************************************************
*
*       _GetIsSelected
*
*/
static U8 _GetIsSelected(void) {
  return 1;
}
/*********************************************************************
*
*       _GetIsPaperEmpty
*
*/
static U8 _GetIsPaperEmpty(void) {
  return 0;
}
/*********************************************************************
*
*       _OnDataReceived
*
*/
static int _OnDataReceived(const U8 * pData, unsigned NumBytes) {
  USB_MEMCPY(_acData, pData, NumBytes);
  _acData[NumBytes] = 0;
  printf(_acData);
  return 0;
}
/*********************************************************************
*
*       _OnReset
*
*/
static void _OnReset(void) {
}
static USB_PRINTER_API _PrinterAPI = {
  _GetDeviceIdString,
  _OnDataReceived,
  _GetHasNoError,
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
  _GetIsSelected,
  _GetIsPaperEmpty,
  _OnReset
};
/*********************************************************************
*
*       Public code
*
**********************************************************************
*/
static const USB_DEVICE_INFO _DeviceInfo = {
  0x8765,                       // VendorId
  0x2114,                       // ProductId, should be unique for this sample
  "Vendor",                     // VendorName
  "Printer",                    // ProductName
  "12345678901234567890"        // SerialNumber
};
/*********************************************************************
*
*       MainTask
*
* Function description
*   USB handling task.
*   Modify to implement the desired protocol
*/
void MainTask(void) {
  USBD_Init();
  USBD_SetDeviceInfo(&_DeviceInfo);
  USB_PRINTER_Init(&_PrinterAPI);
  USBD_Start();
  while (1) {
    //
    // Wait for configuration
    //
    while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED))
           != USB_STAT_CONFIGURED)
    {
      BSP_ToggleLED(0);
      USB_OS_Delay(50);
    }
    //
    // Receive and process data.
    //
    USB_PRINTER_Task();
  }
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

12.3 Target API

This chapter describes the functions and data structures that can be used with the target application.

12.3.1 Interface function list

Function	Description
API functions	
USB_PRINTER_Init()	Initializes the printer module.
USB_PRINTER_Task()	Processes the requests received from the USB Host.
USB_PRINTER_TaskEx()	Processes the requests received from the USB Host.
USB_PRINTER_ConfigIRQProcessing()	Configure printer class to process received data in USB interrupt.
USB_PRINTER_Read()	Reads data from the host.
USB_PRINTER_ReadTimed()	Reads data from the host with a given timeout.
USB_PRINTER_Receive()	Reads data from host.
USB_PRINTER_ReceiveTimed()	Reads data from host with a given time-out.
USB_PRINTER_Write()	Writes data to the host.
USB_PRINTER_WriteTimed()	Writes data to the host within a given timeout.
USB_PRINTER_SetOnVendorRequest()	Sets a callback function that is called when a setup vendor request is sent from the host to the printer.
USB_PRINTER_SetClass()	Sets a custom class/subclass/protocol for the printer class.
Data structures	
USB_PRINTER_API	Initialization structure that is needed when adding a printer interface to emUSB-Device.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2 API functions

12.3.2.1 USB_PRINTER_Init()

Description

Initializes the printer module.

Prototype

```
void USB_PRINTER_Init(USB_PRINTER_API * pAPI);
```

Parameters

Parameter	Description
pAPI	Pointer to an API table that contains all callback functions that are necessary for handling the functionality of a printer.

Additional information

After the initialization of general emUSB-Device, this is the first function that needs to be called when the printer class is used with emUSB-Device.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.2 USB_PRINTER_Task()

Description

Processes the requests received from the USB Host.

Prototype

```
void USB_PRINTER_Task(void);
```

Additional information

This function blocks as long as the USB device is connected to USB host. It handles the requests by calling the functions registered in the call to `USB_PRINTER_Init()`. Do not call this function if you used `USB_PRINTER_ConfigIRQProcessing()`.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.3 USB_PRINTER_TaskEx()

Description

Processes the requests received from the USB Host. Uses overlapped read operation for higher performance.

Prototype

```
void USB_PRINTER_TaskEx(void);
```

Additional information

This function blocks as long as the USB device is connected to USB host. It handles the requests by calling the function registered in the call to `USB_PRINTER_Init()`.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.4 USB_PRINTER_ConfigIRQProcessing()

Description

Configure printer class to process received data in USB interrupt. Must be called after `USB_PRINTER_Init()` and before `USBD_Start()`. After calling this function, `USB_PRINTER_Task()` should never be called.

Prototype

```
void USB_PRINTER_ConfigIRQProcessing(void);
```

Additional information

The printer API function `USB_PRINTER_API -> pfOnDataReceived` is called within the USB interrupt context and must not block.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.5 USB_PRINTER_Read()

Description

Reads data from the host.

Prototype

```
int USB_PRINTER_Read(void     * pData,
                     unsigned   NumBytes);
```

Parameters

Parameter	Description
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Return value

= NumBytes Success.
< NumBytes Error occurred.

Additional information

This function blocks a task until all data has been read. In case of a reset or a disconnect USB_STATUS_ERROR is returned.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.6 USB_PRINTER_ReadTimed()

Description

Reads data from the host with a given timeout.

Prototype

```
int USB_PRINTER_ReadTimed(void     * pData,
                          unsigned   NumBytes,
                          unsigned   ms);
```

Parameters

Parameter	Description
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
ms	Timeout in milliseconds. A zero value results in an infinite timeout.

Return value

= NumBytes	Success.
≥ 0, < NumBytes	Number of bytes that have been read within the given timeout.
< 0	Error.

Additional information

This function blocks a task until all data has been read or a timeout occurs. In case of a reset or a disconnect USB_STATUS_ERROR is returned.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.7 USB_PRINTER_Receive()

Description

Reads data from host. The function blocks until any data has been received. In contrast to USB_PRINTER_Read() this function does not wait for all of NumBytes to be received, but returns after the first packet has been received.

Prototype

```
int USB_PRINTER_Receive(void     * pData,
                        unsigned   NumBytes);
```

Parameters

Parameter	Description
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.

Return value

> 0	Number of bytes that have been read.
= 0	Zero packet received (not every controller supports this!) or the target was disconnected during the function call.
< 0	Error.

Additional information

If no error occurs, this function returns the number of bytes received. Calling USB_PRINTER_Receive() will return as much data as is currently available up to the size of the buffer specified. This function also returns when target is disconnected from host or when a USB reset occurred, it will then return the number of bytes read.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.8 USB_PRINTER_ReceiveTimed()

Description

Reads data from host with a given timeout. The function blocks until any data has been received. In contrast to USB_PRINTER_ReadTimed() this function does not wait for all of NumBytes to be received, but returns after the first packet has been received or after the timeout has been reached.

Prototype

```
int USB_PRINTER_ReceiveTimed(void     * pData,
                             unsigned   NumBytes,
                             unsigned   ms);
```

Parameters

Parameter	Description
pData	Pointer to a buffer where the received data will be stored.
NumBytes	Number of bytes to read.
ms	Timeout in milliseconds.

Return value

> 0	Number of bytes that have been read within the given timeout.
= 0	Zero packet received (not every controller supports this!) or the target was disconnected during the function call.
< 0	An error occurred.

Additional information

If no error occurs, this function returns the number of bytes received. Calling USB_PRINTER_ReceiveTimed() will return as much data as is currently available up to the size of the buffer specified within the specified timeout. This function also returns when target is disconnected from host or when a USB reset occurred, it will then return the number of bytes read.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.9 USB_PRINTER_Write()

Description

Writes data to the host.

Prototype

```
int USB_PRINTER_Write(const void     * pData,
                      unsigned   NumBytes);
```

Parameters

Parameter	Description
pData	Pointer to a buffer that contains the data to be sent.
NumBytes	Number of bytes to write.

Return value

≥ 0 Number of bytes that have been written.
< 0 Error.

Additional information

This function is blocking.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.10 USB_PRINTER_WriteTimed()

Description

Writes data to the host within a given timeout.

Prototype

```
int USB_PRINTER_WriteTimed(const void     * pData,
                           unsigned   NumBytes,
                           int        ms);
```

Parameters

Parameter	Description
pData	Pointer to a buffer that contains the data to be sent.
NumBytes	Number of bytes to write.
ms	Timeout in milliseconds. A zero value results in an infinite timeout. If ms is < 0, the function does not block and may return USB_STATUS_EP_BUSY.

Return value

> 0 Number of bytes that have been written before timeout.
= 0 Timeout occurred.
< 0 Error.

Additional information

If ms ≥ 0, this function blocks the task until all data has been written or a timeout occurred. In case of a reset or a disconnect USB_STATUS_ERROR is returned.

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.11 USB_PRINTER_SetOnVendorRequest()

Description

Sets a callback function that is called when a setup vendor request is sent from the host to the printer. The callback must return "0" if it handled the request and "1" if it did not.

Prototype

```
void USB_PRINTER_SetOnVendorRequest(USB_ON_CLASS_REQUEST * pfOnVendorRequest);
```

Parameters

Parameter	Description
pfOnVendorRequest	Pointer to the callback function.

Additional information

Note that the callback will be called within an ISR, therefore it should never block. If it is necessary to send data from the callback function through endpoint 0, use the function USBD_WriteEP0FromISR().

USB_ON_CLASS_REQUEST is defined as follows:

```
typedef int USB_ON_CLASS_REQUEST(const USB_SETUP_PACKET * pSetupPacket);
```

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.12 USB_PRINTER_SetClass()

Description

Sets a custom class/subclass/protocol for the printer class. Can be used to interface with proprietary manufacturer drivers.

Prototype

```
void USB_PRINTER_SetClass(U8 Class,
                          U8 SubClass,
                          U8 Protocol);
```

Parameters

Parameter	Description
Class	USB class ID overwrite (printer class default is 0x07)
SubClass	USB sub-class ID overwrite (printer class default is 0x01)
Protocol	USB protocol ID overwrite (emUSB-Device printer class default is 0x02)

Additional information

This function must be called after USB_PRINTER_Init() and before USBD_Start().

 © 2010-2026 SEGGER Microcontroller GmbH

12.3.2.13 USB_PRINTER_API

Description

Initialization structure that is needed when adding a printer interface to emUSB-Device. It holds pointers to callback functions the interface invokes when it processes a request from the USB host.

Type definition

```
typedef struct USB_PRINTER_API {
  USB_PRINTER_GET_DEVICE_ID_STRING  * pfGetDeviceIdString;
  USB_PRINTER_ON_DATA_RECEIVED      * pfOnDataReceived;
  USB_PRINTER_GET_HAS_NO_ERROR      * pfGetHasNoError;
  USB_PRINTER_GET_IS_SELECTED       * pfGetIsSelected;
  USB_PRINTER_GET_IS_PAPER_EMPTY    * pfGetIsPaperEmpty;
  USB_PRINTER_ON_RESET              * pfOnReset;
} USB_PRINTER_API;
```

Structure members

Member	Description
pfGetDeviceIdString	The library calls this function when the USB host requests the printer's identification string.
pfOnDataReceived	This function is called when data arrives from the USB host.
pfGetHasNoError	This function should return a non-zero value if the printer has no error.
pfGetIsSelected	This function should return a non-zero value if the printer is selected
pfGetIsPaperEmpty	This function should return a non-zero value if the printer is out of paper.
pfOnReset	The library calls this function if the USB host sends a soft reset command.

Additional information

Detailed information can be found in *USB_PRINTER_API in detail* on page 484.

 © 2010-2026 SEGGER Microcontroller GmbH

12.4 Printer API

This section describes the emUSB-Device Printer API in detail.

12.4.1 General information

The interface includes multiple callback functions which have to be set by the user application. These functions are called by the emUSB-Device stack when the host makes the corresponding inquiries.

 © 2010-2026 SEGGER Microcontroller GmbH

12.4.2 USB_PRINTER_API in detail

12.4.2.1 USB_PRINTER_GET_DEVICE_ID_STRING

Description

The library calls this function when the USB host requests the printer's identification string. This string shall conform to the IEEE 1284 Device ID Syntax.

Type definition

```
typedef const char * USB_PRINTER_GET_DEVICE_ID_STRING(void);
```

Return value

Pointer to the ID string.

Additional information

The return string shall conform to the IEEE 1284 Device ID.

Example

```
"CLASS:PRINTER;
MODEL:HP LaserJet 6MP;
MANUFACTURER:Hewlett-Packard;
DESCRIPTION:Hewlett-Packard LaserJet 6MP Printer;
COMMAND SET:PJL,MLC,PCLXL,PCL,POSTSCRIPT;"
```

 © 2010-2026 SEGGER Microcontroller GmbH

12.4.2.2 USB_PRINTER_ON_DATA_RECEIVED

Description

This function is called when data arrives from USB host.

Type definition

```
typedef int USB_PRINTER_ON_DATA_RECEIVED(const U8     * pData,
                                         unsigned NumBytes);
```

Parameters

Parameter	Description
pData	Pointer to the data.
NumBytes	Data length.

Return value

= 0 More data can be accepted
≠ 0 No more data can be accepted, in this case a stall will be sent back to the host.

 © 2010-2026 SEGGER Microcontroller GmbH

12.4.2.3 USB_PRINTER_GET_HAS_NO_ERROR

Description

This function should return a non-zero value if the printer has no error.

Type definition

```
typedef U8 USB_PRINTER_GET_HAS_NO_ERROR(void);
```

Return value

= 0	Error condition present.
≠ 0	No error.

 © 2010-2026 SEGGER Microcontroller GmbH

12.4.2.4 USB_PRINTER_GET_IS_SELECTED

Description

This function should return a non-zero value if the printer is selected.

Type definition

```
typedef U8 USB_PRINTER_GET_IS_SELECTED(void);
```

Return value

= 0	Not selected.
≠ 0	Selected.

 © 2010-2026 SEGGER Microcontroller GmbH

12.4.2.5 USB_PRINTER_GET_IS_PAPER_EMPTY

Description

This function should return a non-zero value if the printer is out of paper.

Type definition

```
typedef U8 USB_PRINTER_GET_IS_PAPER_EMPTY(void);
```

Return value

= 0	Has paper.
≠ 0	Out of paper.

 © 2010-2026 SEGGER Microcontroller GmbH

12.4.2.6 USB_PRINTER_ON_RESET

Description

The library calls this function if the USB host sends a soft reset command.

Type definition

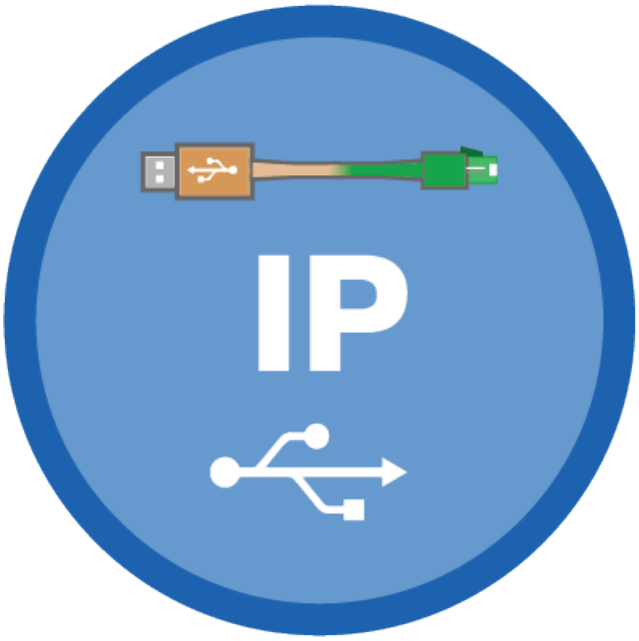
```
typedef void USB_PRINTER_ON_RESET(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 13

IP-over-USB (IP)

This chapter gives a general overview of the IP component and describes how to get the IP component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

13.1 Overview

The IP component is a very convenient package when you need to use IP-based protocols over USB with different host operating systems. It consists of two different components - RNDIS and CDC-ECM combined with the smart capabilities of emUSB-Device-IP to form a cross-platform USB-to-Ethernet device that works on every common Host OS that can handle USB devices.

 © 2010-2026 SEGGER Microcontroller GmbH

13.2 Using only RNDIS or CDC-ECM

The main problem between different Host OSes is that either one IP-over-USB class is supported, which is then not supported on the other Host OS.

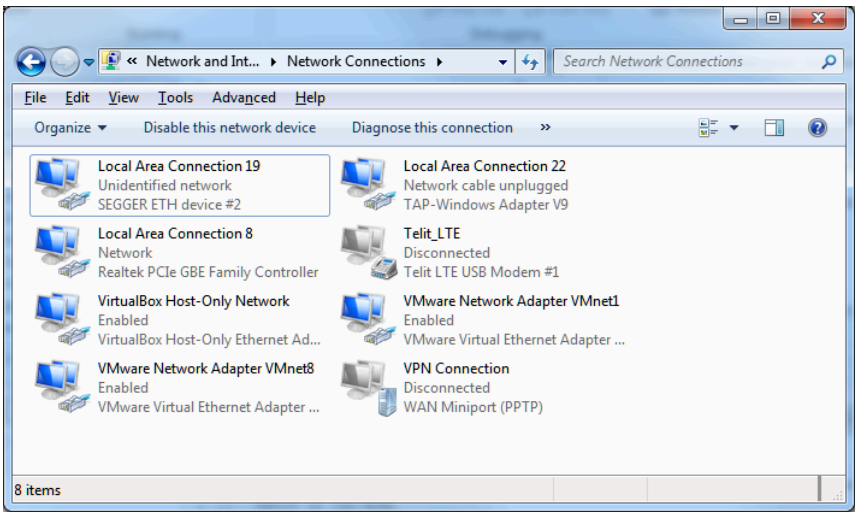
Host OS/Protocol	RNDIS	CDC-ECM
Windows	x	-
Linux	x	x
macOS	-	x
Free/Net/OpenBSD	x	x

Linux and all BSD distributions work with any IP-over-USB interface and therefore can be used with either RNDIS or CDC-ECM. macOS does not support RNDIS, third-party tools may work but are not fully compatible, and when using a new version of macOS, the driver or package may no longer work properly. Windows cannot handle CDC-ECM out of the box. There are third-party drivers which can handle this, but the driver package has to be licensed. Furthermore, a new inf-file needs to be written for your device, and as a consequence, the driver package itself needs to be certified, which involves further costs. Adding new CDC-ECM devices to the inf-file forces you to resign that package once again.

SEGGER's IP-over-USB solution eliminates these limitations.

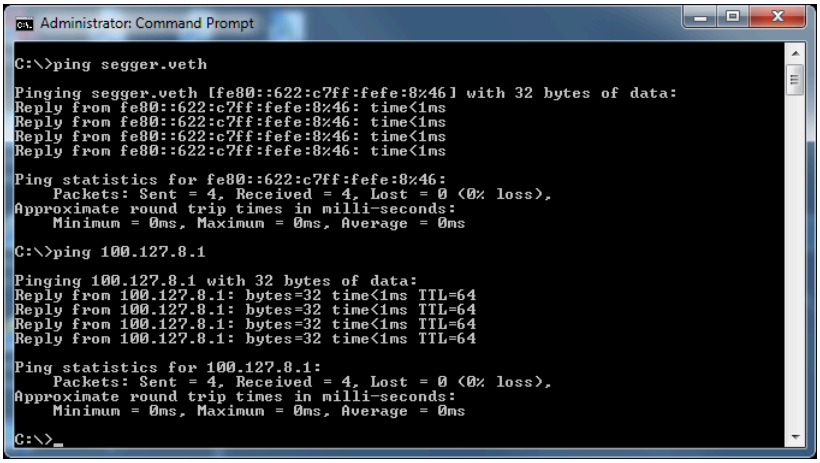
13.2.1 Working with emUSB-Device-IP

An IP-over-USB device connected to a PC running the Windows operating system is listed as a separate network interface in the "Network Connections" window as shown in this screenshot:

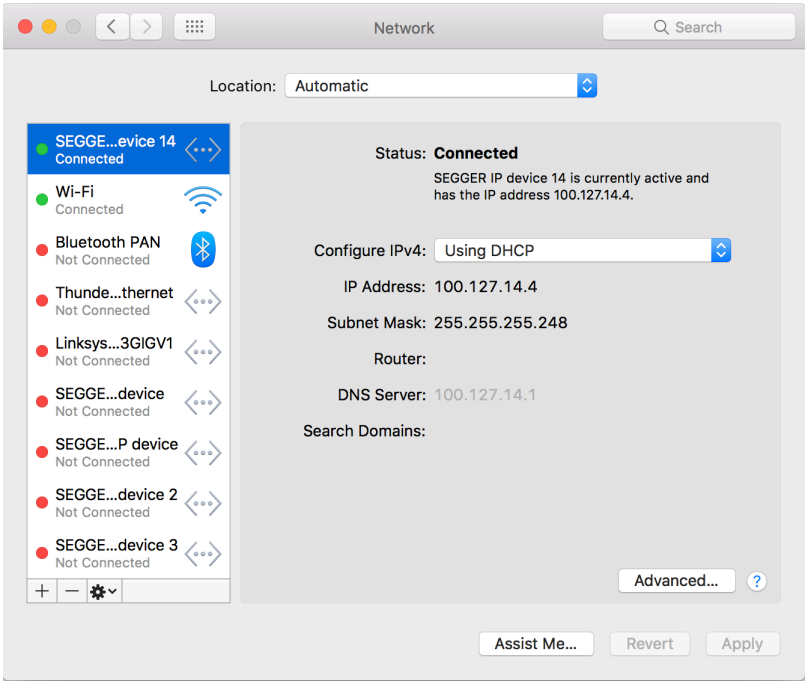


The `ping` command line utility can be used to test the connection to the target as shown below. If the connection is correctly established, the number of lost packets should be 0.

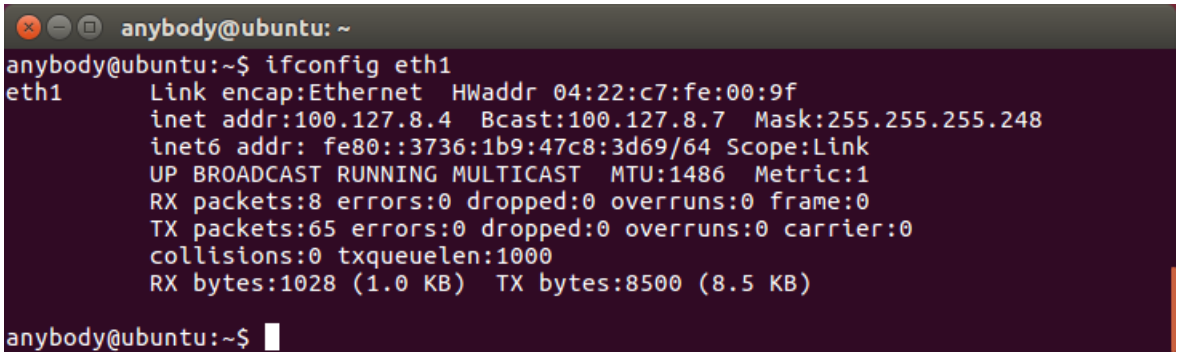
 © 2010-2026 SEGGER Microcontroller GmbH



On macOS IP-over-USB is similarly available:



And on Ubuntu:



 © 2010-2026 SEGGER Microcontroller GmbH

13.3 Configuration

13.3.1 Initial Configuration

To get emUSB-Device-IP up and running as well as to do an initial test, the configuration as delivered should not be modified.

13.3.2 Final configuration

The configuration must only be modified when emUSB-Device-IP is used in your final product. Refer to section *emUSB-Device Configuration* on page 53 to get detailed information about the general emUSB-Device configuration functions which have to be adapted.

> **Note**
>
> Due to an issue in Windows when using IP-over-USB within a multi-interface device, the IP-over-USB interface must be added first. Otherwise, it will not be recognized.

> **Note**
>
> Due to an issue with Windows 7 USB 3.0 drivers, `USBD_EnableIAD()` must be used, even if the device containing the IP-over-USB interface is not a multi-interface device. Otherwise, the device will not be recognized on USB 3.0 ports of a PC running Windows 7.

13.3.3 Class-specific configuration

emUSB-Device-IP specific device information must be provided by the application via the function `USBD_IP_Add()`. A sample of how to use this function can be found in the `IP_Config_IP_over_USB.c`. The file is located in the `Sample\IP` directory of the emUSB-Device shipment. The `IP_Config_IP_over_USB.c` provides a ready-to-use layer and configuration file to be used with *embOS* and *emNet*.

 © 2010-2026 SEGGER Microcontroller GmbH

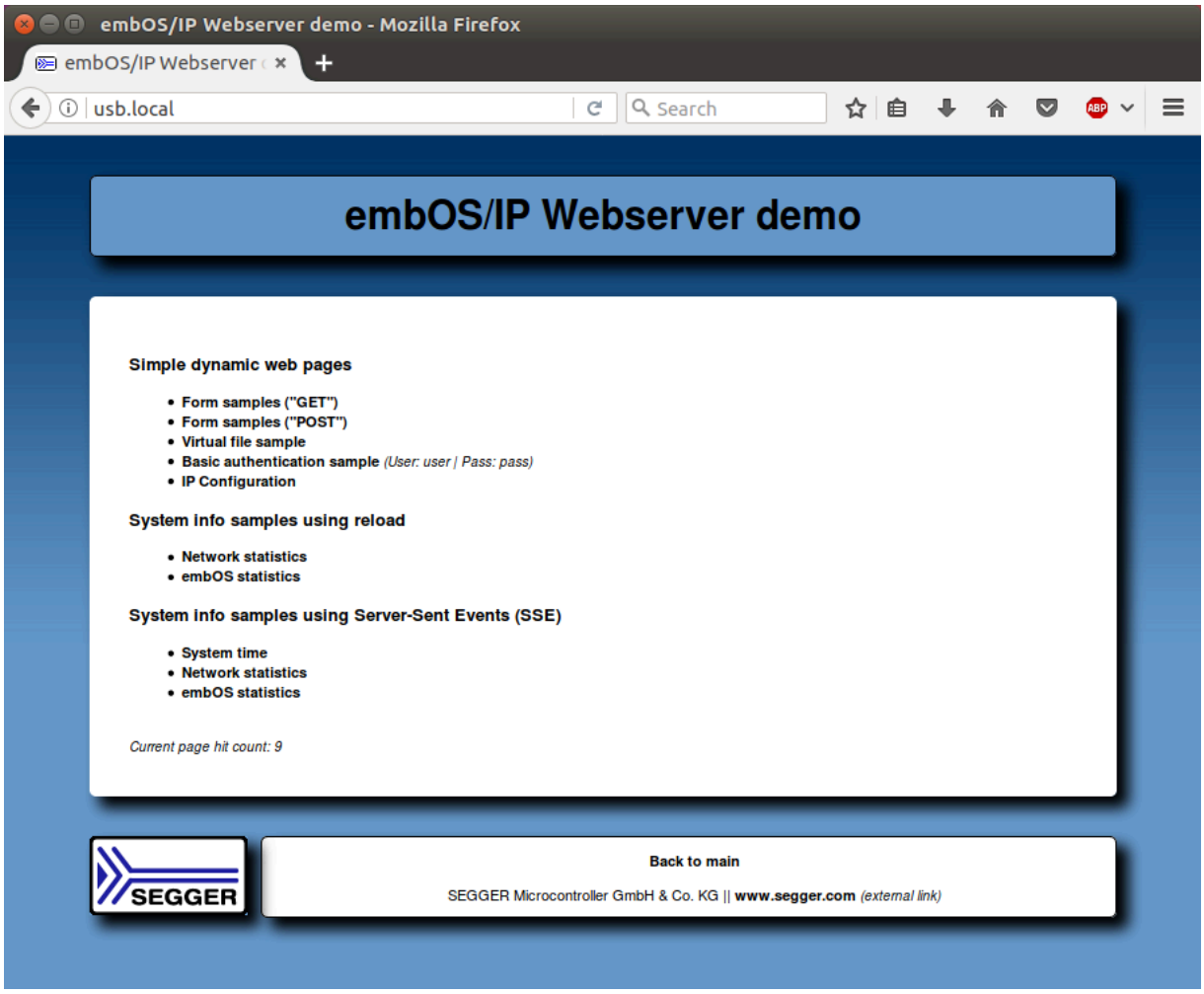
13.4 Running the sample application

The sample application can be found in the `Sample\IP\IP_Config_IP_over_USB.c` file of the emUSB-Device shipment. In order to use the sample application the SEGGER *emNet* middleware component is required. To test the emUSB-Device-IP component any of the *emNet* sample applications can be used in combination with `IP_Config_IP_over_USB.c`. After the sample application is started the USB cable should be connected to the PC and the chosen *emNet* sample can be tested by using the URL “usb.local”. The given `IP_Config_IP_over_USB.c` contains further information on how the interfaces between *emNet* and Host PC are connected. In most cases a DHCP-Server for the PC is necessary. In order to recognize the emNet interface a dedicated name service is used to facilitate the handling. Please refer to the sample for further details.

 © 2010-2026 SEGGER Microcontroller GmbH

13.5 emUSB-Device-IP + emNet as a "USB Webserver"

This method of using emUSB-Device-IP provides a unique customer experience where a USB device can provide a custom web page or any other service through which a customer can interact with the device.



Initially the PC recognizes an RNDIS device. In case of Windows XP and Vista a driver will be necessary (the corresponding inf-file can be found in the `Windows\USB\RNDIS\WinXP_Vista` folder), Windows 7 and above as well as Linux recognize RNDIS automatically. RNDIS from the viewpoint of the PC is a normal Network Interface Controller (NIC) and the PC handles it as such. The default behavior is to request an IP address from a DHCP server. The PC retrieves an IP address from the DHCP server in the device. In our standard sample code the device has the local IP 100.127.<USBAddr>.1 and the PC will get 100.127.<USBAddr>.2 from the DHCP server. With this, the configuration is complete, and the user can access the web interface located on the USB device via the DNS entry, “usb.local”.

 © 2010-2026 SEGGER Microcontroller GmbH

13.6 Target API

Function	Description
API functions	
USBD_IP_Add()	Adds support for the IP component to USB stack.
USBD_IP_Task()	Obsolete.
Data structures	
USB_IP_INIT_DATA	Structure which stores the parameters of the IP component.

 © 2010-2026 SEGGER Microcontroller GmbH

13.6.1 API functions

13.6.1.1 USBD_IP_Add()

Description

Adds support for the IP component to USB stack. Internally CDC-ECM and RNDIS is initialized. The IP component switches automatically between the two.

Prototype

```
void USBD_IP_Add(const USB_IP_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a filled USB_IP_INIT_DATA structure data.

 © 2010-2026 SEGGER Microcontroller GmbH

13.6.1.2 USBD_IP_Task()

Description

Obsolete. Returns when USB is disconnected.

Prototype

```
void USBD_IP_Task(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

13.6.2 Data structures

13.6.2.1 USB_IP_INIT_DATA

Description

Structure which stores the parameters of the IP component.

Type definition

```
typedef struct USB_IP_INIT_DATA {
  U8                              EPIn;
  U8                              EPOut;
  U8                              EPInt;
  const USB_IP_NI_DRIVER_API    * pDriverAPI;
  USB_IP_NI_DRIVER_DATA           DriverData;
  const USB_RNDIS_DEVICE_INFO   * pRndisDevInfo;
} USB_IP_INIT_DATA;
```

Structure members

Member	Description
EPIn	Bulk IN endpoint to send data packets to the USB host.
EPOut	Bulk OUT endpoint to receive data packets from the USB host.
EPInt	Interrupt IN endpoint to send notifications to the USB host.
pDriverAPI	Network interface driver API.
DriverData	Data passed at initialization to low-level driver.
pRndisDevInfo	Pointer to a filled USB_RNDIS_DEVICE_INFO structure.

Additional information

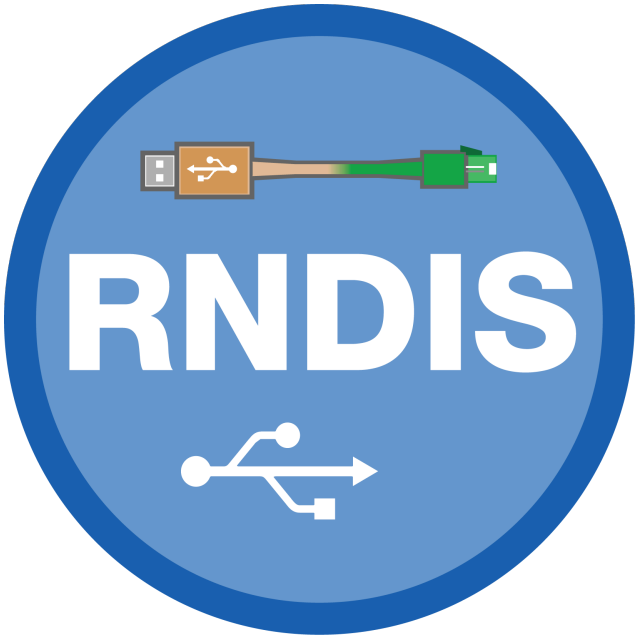
This structure holds the endpoints that should be used with the IP component. Refer to USBD_AddEP() for more information about how to add an endpoint.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 14

Remote NDIS (RNDIS)

This chapter gives a general overview of the Remote Network Driver Interface Specification class and describes how to get the RNDIS component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

14.1 Overview

The Remote Network Driver Interface Specification (RNDIS) is a Microsoft proprietary USB class protocol which can be used to create a virtual Ethernet connection between a USB device and a host PC. A TCP/IP stack like *emNet* is required on the USB device side to handle the actual IP communication. Any available IP protocol (UDP, TCP, FTP, HTTP, etc.) can be used to exchange data. On a typical Cortex-M CPU running at 120 MHz, a transfer speed of about 5 MB/s can be achieved when using a high-speed USB connection.

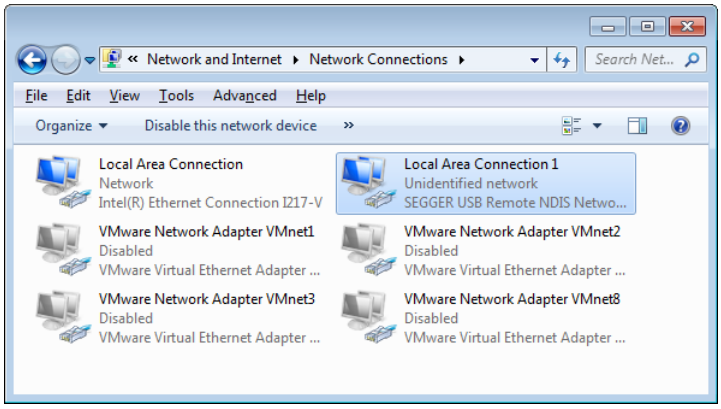
USB RNDIS is supported by all Windows operating systems starting with Windows XP, as well as by Linux with kernel versions newer than 2.6.34. An .inf file is required for the installation on Microsoft Windows systems older than Windows 7. The emUSB-Device-RNDIS package includes `.inf` files for Windows versions older than Windows 7. macOS will require a third-party driver to work with RNDIS, which can be downloaded from here: https://joshuawise.com/horndis which will only work for certain macOS versions.

emUSB-Device-RNDIS contains the following components:

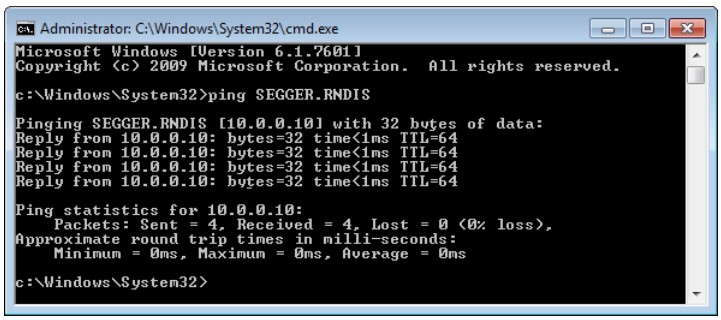
- Generic USB handling
- RNDIS device class implementation
- Network interface driver which uses *emNet* as TCP/IP stack.
- A sample application demonstrating how to work with RNDIS.

14.1.1 Working with RNDIS

Any USB RNDIS device connected to a PC running the Windows operating system is listed as a separate network interface in the “Network Connections” window as shown in this screenshot:



The `ping` command line utility can be used to test the connection to the target as shown below. If the connection is correctly established, the number of lost packets should be 0.



14.1.2 Additional information

More technical details about RNDIS can be found here:

https://docs.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-remote-ndis--rndis-

 © 2010-2026 SEGGER Microcontroller GmbH

14.2 Configuration

14.2.1 Initial Configuration

To get emUSB-Device-RNDIS up and running as well as to perform an initial test, the configuration as delivered should not be modified.

14.2.2 Final configuration

The configuration must only be modified when emUSB-Device is used in your final product. Refer to section *emUSB-Device Configuration* on page 53 to get detailed information about the general emUSB-Device configuration functions which have to be adapted.

> **Note**
>
> Due to an issue on Windows when using RNDIS within a multi-interface device, the RNDIS interface must be added first. Otherwise, it will not be recognized.

> **Note**
>
> Due to an issue with Windows 7 USB 3.0 drivers, `USBD_EnableIAD()` must be used, even if the device containing the RNDIS interface is not a multi-interface device. Otherwise, the device will not be recognized on USB 3.0 ports of a PC running Windows 7.

14.2.3 Class-specific configuration

RNDIS-specific device information must be provided by the application via the function `USBD_RNDIS_SetDeviceInfo()` before the USB stack is started using `USBD_Start()`. A sample of how to use this function can be found in the `IP_Config_RNDIS.c`. The file is located in the `Sample\RNDIS` directory of the emUSB-Device shipment. The `IP_Config_RNDIS.c` provides a ready-to-use layer and configuration file to be used with *embOS* and *emNet*.

 © 2010-2026 SEGGER Microcontroller GmbH

14.3 Running the sample application

The sample application can be found in the `Sample\RNDIS\IP_Config_RNDIS.c` file of the emUSB-Device shipment. In order to use the sample application the SEGGER *emNet* middleware component is required. To test the emUSB-Device-RNDIS component any of the *emNet* sample applications can be used in combination with `IP_Config_RNDIS.c`. After the sample application is started the USB cable should be connected to the PC and the chosen *emNet* sample can be tested using the appropriate methods.

14.3.1 IP_Config_RNDIS.c in detail

The main part of the sample application is implemented in the function MainTask() which runs as an independent task.

```
// _Connect() - excerpt from IP_Config_RNDIS.c
static int _Connect(unsigned IFaceId) {
  U32 ServerIpAddr;
  U32 SubnetMask;

  ServerIpAddr = IP_BYTES2ADDR(10, 0, 0, 10);
  SubnetMask = IP_BYTES2ADDR(0xff,0xff,0xff,0xf8);
  IP_SetAddrMaskEx(IFaceId, ServerIpAddr, SubnetMask);
  IP_DHCPS_ConfigPool(IFaceId, ServerIpAddr + 1, SubnetMask, 4);
  // Setup IP pool to distribute.
  IP_DHCPS_ConfigDNSAddr(IFaceId, &ServerIpAddr, 1);
  IP_DHCPS_Init(IFaceId);
  IP_DHCPS_Start(IFaceId);
  IP_NETBIOS_Init(IFaceId, _aNetNames, 0);
  // Init NetBIOS.
  IP_NETBIOS_Start(IFaceId);
  // Start NetBIOS.
  USBD_Init();
  USBD_SetDeviceInfo(&USB_DeviceInfo);
  USBD_RNDIS_SetDeviceInfo(&USB_RNDIS_DeviceInfo);
  //
  // Although we do not have a composite device, we enable IAD as a workaround
  // for the buggy Intel USB driver on Windows 7
  //
  USBD_EnableIAD();
  _AddRNDIS();
  USBD_Start();
  return 0;                   // Successfully connected.
}
```

The first step is to initialize the DHCP server component which assigns the IP address to the PC side. The target is configured with the IP address 10.0.0.10. The DHCP server is configured to distribute IP addresses starting from 10.0.0.11, therefore the PC will receive the IP address 10.0.0.11. Then the USB stack is initialized and the RNDIS interface is added to it. The function `_AddRNDIS()` configures all required endpoints.

```
// _AddRNDIS() - excerpt from IP_Config_RNDIS.c
static U8 _abReceiveBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
static void _AddRNDIS(void) {
  USB_RNDIS_INIT_DATA InitData;

  memset(&InitData, 0, sizeof(InitData));
  InitData.EPOut      = USBD_AddEP(USB_DIR_OUT,
                                   USB_TRANSFER_TYPE_BULK,
                                   0,
                                   _abReceiveBuffer, sizeof(_abReceiveBuffer));
  InitData.EPIn       = USBD_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_BULK,
                                   0, NULL, 0);
  InitData.EPInt      = USBD_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_INT,
                                   5, NULL, 0);
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
  InitData.pDriverAPI = &USB_Driver_IP_NI;
  InitData.DriverData.pDriverData = (void *)_IFaceId;
  USBD_RNDIS_Add(&InitData);
}
```

The size of the `_abReceiveBuffer` buffer must be a multiple of the USB max packet size. `USB_Driver_IP_NI` is the network interface driver which implements the connection to the IP stack. Optionally, a HW address may be configured here, which is assigned to the PC network interface. If not set (`pHWAddr = NULL`), the HW address is generated automatically later while setting the interface up.

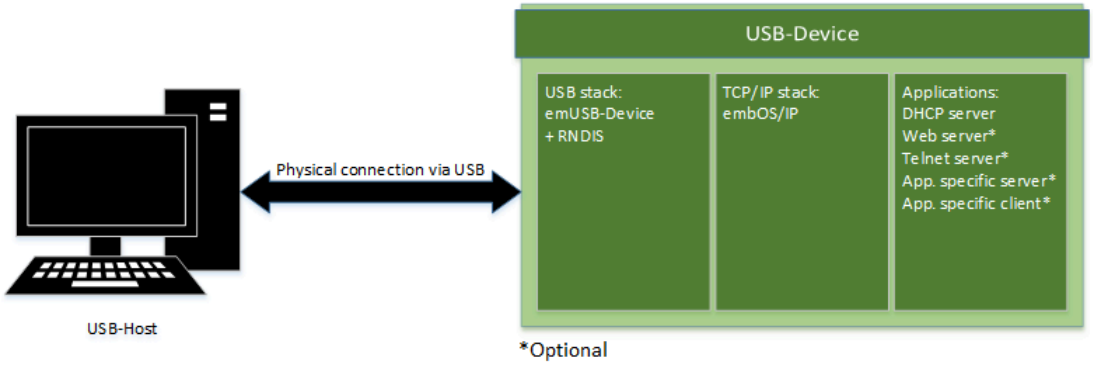
The IP stack is configured to use the network interface driver of emUSB-Device-RNDIS. For more information about the configuration of the IP stack, refer to the *emNet* manual.

```
// IP_X_Config() - excerpt from IP_Config.c
#include "USB_Driver_IP_NI.h"
void IP_X_Config(void) {
  <...>
  //
  // Add and configure the RNDIS driver.
  // The local IP address is 10.0.0.10/8.
  //
  IFaceId = IP_AddEtherInterface(&USB_IP_Driver);
  IP_SetIFaceConnectHook(IFaceId, _Connect);
  IP_SetIFaceDisconnectHook(IFaceId, _Disconnect);
  _IFaceId = IFaceId;
  <...>
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

14.4 RNDIS + emNet as a "USB Web Server"

This method of using RNDIS provides a unique customer experience where a USB device can provide a custom web page or any other service through which a customer can interact with the device.



Initially the PC recognizes an RNDIS device. In case of Windows XP and Vista, a driver will be necessary; Windows 7 and above as well as Linux recognize RNDIS automatically. RNDIS from the viewpoint of the PC is a normal Network Interface Controller (NIC) and the PC handles it as such. The default behavior is to request an IP address from a DHCP server. The PC retrieves an IP address from the DHCP server in the device. In our standard sample code, the device has the local IP 10.0.0.10 and the PC will get 10.0.0.11 from the DHCP server. With this, the configuration is complete, and the user can access the web interface located on the USB device via 10.0.0.10. To improve the ease of use, NetBIOS can be used to give the device an easily readable name.

 © 2010-2026 SEGGER Microcontroller GmbH

14.5 Target API

Function	Description
API functions	
USBD_RNDIS_Add()	Adds an RNDIS class interface to the USB stack.
USBD_RNDIS_Task()	Obsolete.
USBD_RNDIS_SetDeviceInfo()	Provides device information for USB enumeration to the stack.
Data structures	
USB_RNDIS_INIT_DATA	Structure which stores the parameters of the RNDIS interface.
USB_RNDIS_DEVICE_INFO	Device information that must be provided by the application via the function USBD_RNDIS_SetDeviceInfo() before the USB stack is started using USBD_Start().
USB_IP_NI_DRIVER_API	This structure contains the callback functions for the network interface driver.
USB_IP_NI_DRIVER_DATA	Configuration data passed to network interface driver at initialization.

 © 2010-2026 SEGGER Microcontroller GmbH

14.5.1 API functions

14.5.1.1 USBD_RNDIS_Add()

Description

Adds an RNDIS class interface to the USB stack.

Prototype

```
void USBD_RNDIS_Add(const USB_RNDIS_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to initialization data.

Additional information

This function should be called after the USB core has been initialized to add an RNDIS interface to emUSB-Device. The initialization data is passed to the function in the structure pointed to by pInitData. Refer to USB_RNDIS_INIT_DATA for more information.

 © 2010-2026 SEGGER Microcontroller GmbH

14.5.1.2 USBD_RNDIS_Task()

Description

Obsolete. Returns when USB is disconnected.

Prototype

```
void USBD_RNDIS_Task(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

14.5.1.3 USBD_RNDIS_SetDeviceInfo()

Description

Provides device information for USB enumeration to the stack.

Prototype

```
void USBD_RNDIS_SetDeviceInfo(const USB_RNDIS_DEVICE_INFO * pDeviceInfo);
```

Parameters

Parameter	Description
pDeviceInfo	Pointer to a USB_RNDIS_DEVICE_INFO structure containing the device information. Must point to static data that remains unchanged while the stack is running.

 © 2010-2026 SEGGER Microcontroller GmbH

14.5.2 Data structures

14.5.2.1 USB_RNDIS_INIT_DATA

Description

Structure which stores the parameters of the RNDIS interface.

Type definition

```
typedef struct USB_RNDIS_INIT_DATA {
  U8                              EPIn;
  U8                              EPOut;
  U8                              EPInt;
  const USB_IP_NI_DRIVER_API   * pDriverAPI;
  USB_IP_NI_DRIVER_DATA          DriverData;
  unsigned                       DataInterfaceNo;
} USB_RNDIS_INIT_DATA;
```

Structure members

Member	Description
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host.
EPInt	Interrupt IN endpoint for sending status information.
pDriverAPI	Pointer to the Network interface driver API.
DriverData	Configuration data for the network interface driver.
DataInterfaceNo	Internal use.

Additional information

This structure holds the endpoints that should be used by the RNDIS interface (EPin, EPOut and EPInt). Refer to USBD_AddEP() for more information about how to add an endpoint.

 © 2010-2026 SEGGER Microcontroller GmbH

14.5.2.2 USB_RNDIS_DEVICE_INFO

Description

Device information that must be provided by the application via the function USBD_RNDIS_SetDeviceInfo() before the USB stack is started using USBD_Start().

Type definition

```
typedef struct {
  U32     VendorId;
  char  * sDescription;
  U16     DriverVersion;
} USB_RNDIS_DEVICE_INFO;
```

Structure members

Member	Description
VendorId	A 24-bit Organizationally Unique Identifier (OUI) of the vendor. This is the same value as the one stored in the first 3 bytes of a HW (MAC) address. Only the least significant 24 bits of the returned value are used.
sDescription	0-terminated ASCII string describing the device. The string is then sent to the host system.
DriverVersion	16-bit value representing the firmware version. The high-order byte specifies the major version and the low-order byte the minor version.

 © 2010-2026 SEGGER Microcontroller GmbH

14.5.3 Driver interface

14.5.3.1 USB_IP_NI_DRIVER_API

Description

This structure contains the callback functions for the network interface driver.

Type definition

```
typedef struct {
  USB_IP_NI_INIT                      * pfInit;
  USB_IP_NI_GET_PACKET_BUFFER         * pfGetPacketBuffer;
  USB_IP_NI_WRITE_PACKET              * pfWritePacket;
  USB_IP_NI_SET_PACKET_FILTER         * pfSetPacketFilter;
  USB_IP_NI_GET_LINK_STATUS           * pfGetLinkStatus;
  USB_IP_NI_GET_LINK_SPEED            * pfGetLinkSpeed;
  USB_IP_NI_GET_HWADDR                * pfGetHWAddr;
  USB_IP_NI_GET_STATS                 * pfGetStats;
  USB_IP_NI_GET_MTU                   * pfGetMTU;
  USB_IP_NI_RESET                     * pfReset;
  USB_IP_NI_SET_WRITE_PACKET_FUNC     * pfSetWritePacketFunc;
  USB_IP_NI_SET_REPORT_LINKSTATE_FUNC * pfSetReportLinkstateFunc;
} USB_IP_NI_DRIVER_API;
```

Structure members

Member	Description
pfInit	Initializes the driver.
pfGetPacketBuffer	Returns a buffer for a data packet.
pfWritePacket	Delivers a data packet to target IP stack.
pfSetPacketFilter	Configures the type of accepted data packets.
pfGetLinkStatus	Returns the status of the connection to target IP stack.
pfGetLinkSpeed	Returns the connection speed.
pfGetHWAddr	Returns the HW address of the PC.
pfGetStats	Returns statistical counters.
pfGetMTU	Returns the size of the largest data packet which can be transferred.
pfReset	Resets the driver.
pfSetWritePacketFunc	Allows to change the WritePacket callback which was set by pfInit.
pfSetReportLinkstate-Func	Allows to set the report link state change.

Additional information

The emUSB-Device-RNDIS/emUSB-Device-CDC-ECM component calls the functions of this API to exchange data and status information with the IP stack running on the target.

 © 2010-2026 SEGGER Microcontroller GmbH

14.5.3.2 USB_IP_NI_DRIVER_DATA

Description

Configuration data passed to network interface driver at initialization.

Type definition

```
typedef struct USB_IP_NI_DRIVER_DATA {
  const U8   * pHWAddr;
  unsigned     NumBytesHWAddr;
  void       * pDriverData;
  void       * pUSBContext;
} USB_IP_NI_DRIVER_DATA;
```

Structure members

Member	Description
pHWAddr	Optional pointer to a HW address (or MAC address) of the host network interface.
NumBytesHWAddr	Number of bytes in the HW address. Typically 6 bytes.
pDriverData	Pointer to a user context.
pUSBContext	Internal use.

Additional information

When pHWAddr is NULL the MAC is automatically generated.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6 RNDIS IP Driver

This section describes the emUSB-Device RNDIS IP stack interface in detail.

14.6.1 General information

This release comes with an IP NI driver which uses emNet as the IP stack. If you are using emNet, this chapter can be ignored. This chapter is for those who wish to write their own IP stack interface for a third-party IP stack.

The IP interface is handled through an API-table, which contains all relevant functions necessary for read/write operations and initialization.

14.6.2 Interface function list

As described above, access to network functions is realized through an API-function table of type `USB_IP_NI_DRIVER_API`. The structure is declared in `USB_Driver_IP_NI.h`, and it is described in section *Data structures* on page 500.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3 USB_IP_NI_DRIVER_API in detail

14.6.3.1 USB_IP_NI_INIT

Description

Initializes the driver.

Type definition

```
typedef unsigned (USB_IP_NI_INIT)(const USB_IP_NI_DRIVER_DATA * pDriverData,
                                  USB_IP_WRITE_PACKET   * pfWritePacket);
```

Parameters

Parameter	Description
pDriverData	in Pointer to driver configuration data.
pfWritePacket	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

Return value

IP NI driver instance ID.

Additional information

This function is called when the RNDIS/ECM interface is added to the USB stack. Typically the function makes a local copy of the HW address passed in the pDriverData structure. For more information this structure refer to USB_IP_NI_DRIVER_DATA.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.2 USB_IP_NI_GET_PACKET_BUFFER

Description

Returns a buffer for a data packet.

Type definition

```
typedef void * (USB_IP_NI_GET_PACKET_BUFFER)(unsigned Id,
                                             unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
NumBytes	Size of the requested buffer in bytes.

Return value

≠ NULL Pointer to allocated buffer
= NULL No buffer available

Additional information

The function should allocate a buffer of the requested size. If the buffer can not be allocated a NULL pointer should be returned. The function is called when a data packet is received from PC. The packet data is stored in the returned buffer.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.3 USB_IP_NI_WRITE_PACKET

Description

Delivers a data packet to target IP stack.

Type definition

```
typedef void (USB_IP_NI_WRITE_PACKET)(      unsigned Id,
                                      const void   * pData,
                                            unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pData	in Data of the received packet.
NumBytes	Number of bytes stored in the buffer.

Additional information

The function is called after a data packet has been received from USB. pData points to the buffer returned by the USB_IP_NI_GET_PACKET_BUFFER function.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.4 USB_IP_NI_SET_PACKET_FILTER

Description

Configures the type of accepted data packets.

Type definition

```
typedef void (USB_IP_NI_SET_PACKET_FILTER)(unsigned Id,
                                           U32      Mask);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
Mask	Type of accepted data packets.

Additional information

The Mask parameter should be interpreted as a boolean value. A value different than 0 indicates that the connection to target IP stack should be established. When the function is called with the Mask parameter set to 0 the connection to target IP stack should be interrupted.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.5 USB_IP_NI_GET_LINK_STATUS

Description

Returns the status of the connection to target IP stack.

Type definition

```
typedef int (USB_IP_NI_GET_LINK_STATUS)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

Return value

USB_IP_NI_LINK_STATUS_DISCONNECTED	Not connected to target IP stack.
USB_IP_NI_LINK_STATUS_CONNECTED	Connected to target IP stack.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.6 USB_IP_NI_GET_LINK_SPEED

Description

Returns the connection speed.

Type definition

```
typedef U32 (USB_IP_NI_GET_LINK_SPEED)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

Return value

≠ 0	The connection speed in units of 100 bits/sec (between the USB_IP_NI module and the target IP stack).
= 0	Not connected to the target IP stack.

Additional information

In this implementation the return value is 0 when the instance Id was not found or 1000000 (100 Mbit/s).

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.7 USB_IP_NI_GET_HWADDR

Description

Returns the HW address (MAC address) of the host network interface (PC).

Type definition

```
typedef void (USB_IP_NI_GET_HWADDR)(unsigned Id,
                                     U8     * pAddr,
                                     unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pAddr	out The HW address.
NumBytes	Maximum number of bytes to store into pAddr.

Additional information

The returned HW address is the one passed to the driver in the call to USB_IP_NI_INIT. Typically the HW address is 6 bytes long.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.8 USB_IP_NI_GET_STATS

Description

Returns statistical counters.

Type definition

```
typedef U32 (USB_IP_NI_GET_STATS)(unsigned Id,
                                  int      Type);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
Type	The type of information requested. See table below.

Return value

Value of the requested statistical counter.

Additional information

The counters should be set to 0 when the USB_IP_NI_RESET function is called.

Permitted values for parameter Type	
USB_IP_NI_STATS_CLEAR_ALL_STATS	Special type which will instruct the module to reset all statistical counters to zero.
USB_IP_NI_STATS_WRITE_PACKET_OK	Number of packets sent without errors to target IP stack.
USB_IP_NI_STATS_WRITE_PACKET_ERROR	Number of packets sent with errors to target IP stack.
USB_IP_NI_STATS_READ_PACKET_OK	Number of packets received without errors from target IP stack.
USB_IP_NI_STATS_READ_PACKET_ERROR	Number of packets received with errors from target IP stack.
USB_IP_NI_STATS_READ_NO_BUFFER	Number of packets received from target IP stack but dropped.
USB_IP_NI_STATS_READ_ALIGN_ERROR	Number of packets received from target IP stack with alignment errors.
USB_IP_NI_STATS_WRITE_ONE_COLLISION	Number of packets which were not sent to target IP stack due to the occurrence of one collision.
USB_IP_NI_STATS_WRITE_MORE_COLLISIONS	Number of packets which were not sent to target IP stack due to the occurrence of one or more collisions.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.9 USB_IP_NI_GET_MTU

Description

Returns the maximum transmission unit, the size of the largest data packet which can be transferred.

Type definition

```
typedef U32 (USB_IP_NI_GET_MTU)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

Return value

The MTU size in bytes. Typically 1500 bytes.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.10 USB_IP_NI_RESET

Description

Resets the driver.

Type definition

```
typedef void (USB_IP_NI_RESET)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.11 USB_IP_NI_SET_WRITE_PACKET_FUNC

Description

Changes the USB_IP_WRITE_PACKET callback which was added via USB_IP_NI_INIT to a different callback function. This function is only called by the stack when IP-over-USB is used. It is not called when RNDIS or ECM is used standalone.

Type definition

```
typedef void (USB_IP_NI_SET_WRITE_PACKET_FUNC)
                                          (unsigned              Id,
                                           USB_IP_WRITE_PACKET * pfWritePacket);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pfWritePacket	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

 © 2010-2026 SEGGER Microcontroller GmbH

14.6.3.12 USB_IP_NI_SET_REPORT_LINKSTATE_FUNC

Description

Changes the USB_IP_REPORT_LINKSTATE callback. Normally this is called only once per initialization in order to allow to send notification to the host that the link state has been changed.

Type definition

```
typedef void (USB_IP_NI_SET_REPORT_LINKSTATE_FUNC)
                                    (unsigned                   Id,
                                     USB_IP_REPORT_LINKSTATE * pfReportLinkState);
```

Parameters

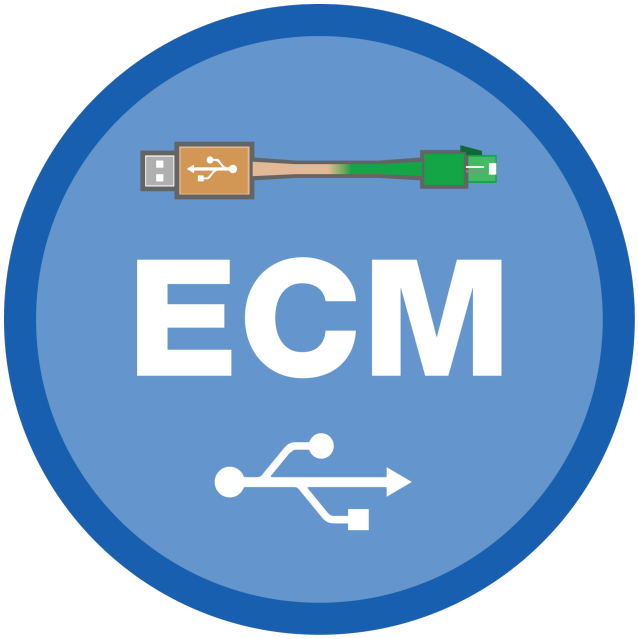
Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pfReportLinkState	Callback function called by the IP stack to notify the host that the link state has been changed.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 15

CDC-ECM

This chapter gives a general overview of the Communications Device Class / Ethernet Control Model class and describes how to get the ECM component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

15.1 Overview

The Communications Device Class / Ethernet Control Model is a USB class protocol of the USB Implementers Forum which can be used to create a virtual Ethernet connection between a USB device and a host PC. A TCP/IP stack like *emNet* is required on the USB device side to handle the actual IP communication. Any available IP protocol (UDP, TCP, FTP, HTTP, etc.) can be used to exchange data.

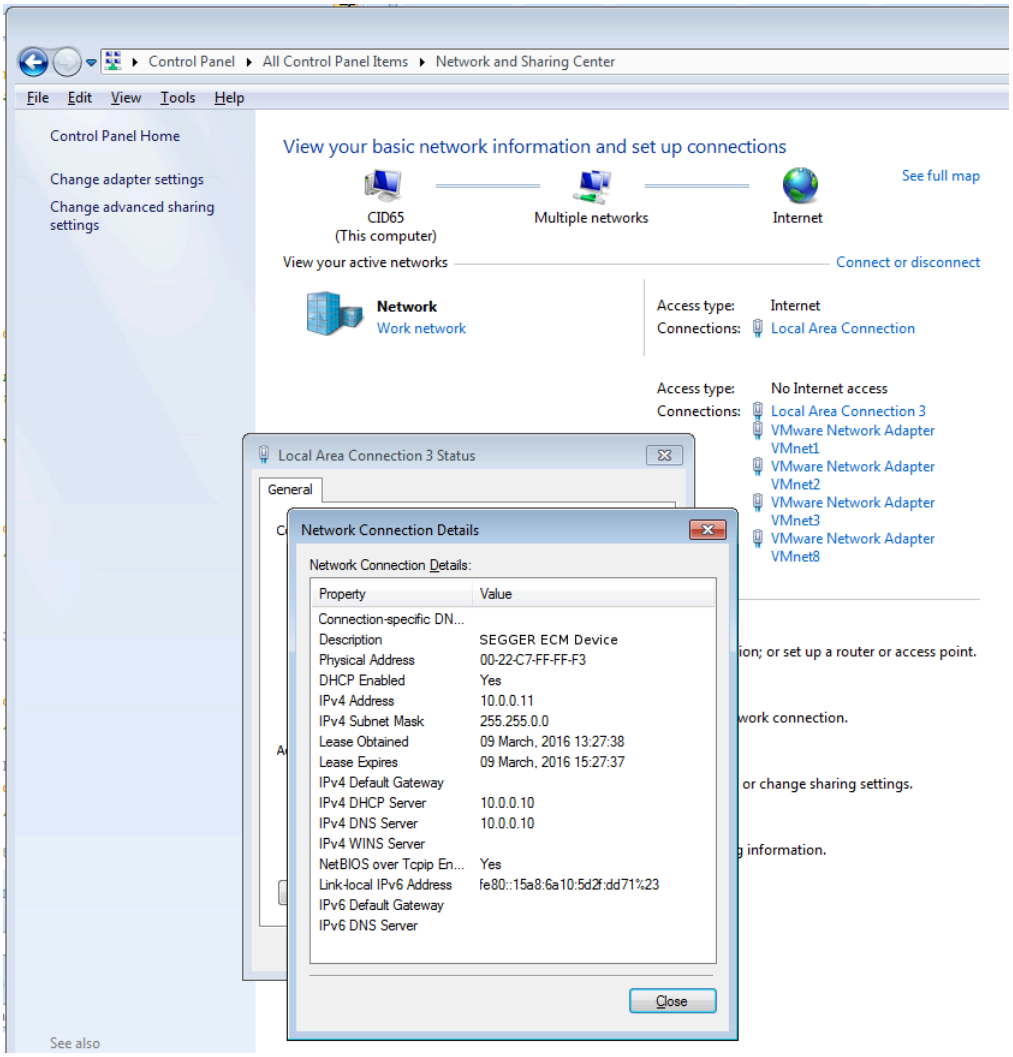
USB ECM is supported by the Linux operating system. To use it on Windows, a third-party driver (not contained in emUSB-Device-ECM) has to be installed on the Windows system.

emUSB-Device-ECM contains the following components:

- Generic USB handling
- ECM device class implementation
- Network interface driver which uses *emNet* as a TCP/IP stack.
- A sample application demonstrating how to work with ECM.

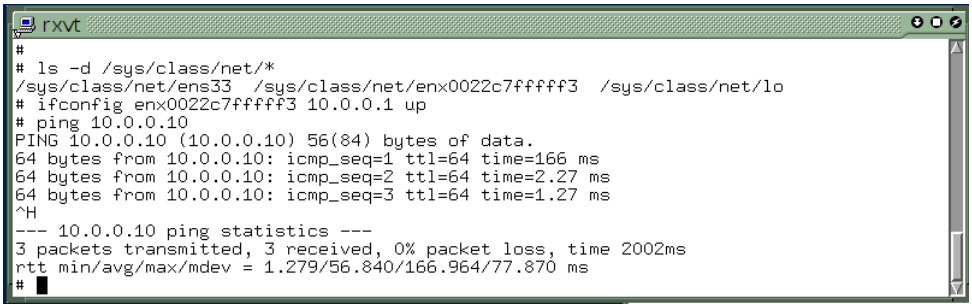
15.1.1 Working with CDC-ECM

Any USB ECM device connected to a PC running the Windows operating system (with a third-party CDC-ECM driver installed) is listed as a separate network interface in the “Network Connections” window as shown in this screenshot:



The `ping` command line utility can be used to test the connection to the target as shown below. If the connection is correctly established, the number of lost packets should be 0. The following screenshot shows a manual configuration and ping on Linux.

 © 2010-2026 SEGGER Microcontroller GmbH



15.1.2 Additional information

More technical details about CDC-ECM can be found on https://www.usb.org in the Class definitions for Communication Devices 1.2 package: CDC Subclass for Ethernet Control Model Devices.

 © 2010-2026 SEGGER Microcontroller GmbH

15.2 Configuration

15.2.1 Initial configuration

To get emUSB-Device-ECM up and running as well as to do an initial test, the configuration as delivered should not be modified. When using on Windows with a third-party driver, the vendor ID and product ID must match the IDs configured in the `.inf` file of the driver.

15.2.2 Final configuration

The configuration must only be modified when emUSB-Device is used in your final product. Refer to section *emUSB-Device Configuration* on page 53 to get detailed information about the general emUSB-Device configuration functions which have to be adapted.

 © 2010-2026 SEGGER Microcontroller GmbH

15.3 Running the sample application

The sample application can be found in the `Sample\ECM\IP_Config_ECM.c` file of the emUSB-Device shipment. In order to use the sample application, the SEGGER *emNet* middleware component is required. To test the emUSB-Device-ECM component, any of the *emNet* sample applications can be used in combination with `IP_Config_ECM.c`. After the sample application is started, the USB cable should be connected to the PC and the chosen *emNet* sample can be tested using the appropriate methods.

15.3.1 IP_Config_ECM.c in detail

The main part of the sample application is implemented in the function `MainTask()` which runs as an independent task.

```
// _Connect() - excerpt from IP_Config_ECM.c
static int _Connect(unsigned IFaceId) {
  U32 Server = IP_BYTES2ADDR(10, 0, 0, 10);
  IP_DHCPS_ConfigPool(IFaceId, IP_BYTES2ADDR(10, 0, 0, 11), 0xFF000000, 20);
  IP_DHCPS_ConfigDNSAddr(IFaceId, &Server, 1);
  IP_DHCPS_Init(IFaceId);
  IP_DHCPS_Start(IFaceId);
  USBD_Init();
  USBD_SetDeviceInfo(&USB_DeviceInfo);
  _AddECM();
  USBD_Start();
  return 0;                   // Successfully connected.
}
```

The first step is to initialize the DHCP server component which assigns the IP address for the PC side. The target is configured with the IP address 10.0.0.10. The DHCP server is configured to distribute IP addresses starting from 10.0.0.11, therefore the PC will receive the IP address 10.0.0.11. Then the USB stack is initialized and the ECM interface is added to it. The function `_AddECM()` configures all required endpoints and sets the HW address of the PC network interface.

```
// _AddECM() - excerpt from IP_Config_ECM.c
static U8 _abReceiveBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
static void _AddECM(void) {
  USB_ECM_INIT_DATA InitData;
  InitData.EPOut      = USBD_AddEP(USB_DIR_OUT,
                                   USB_TRANSFER_TYPE_BULK,
                                   0,
                                   _abReceiveBuffer, sizeof(_abReceiveBuffer));
  InitData.EPIn       = USBD_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_BULK,
                                   0, NULL, 0);
  InitData.EPInt      = USBD_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_INT,
                                   32, NULL, 0);
  InitData.pDriverAPI = &USB_Driver_IP_NI;
  InitData.DriverData.pDriverData    = (void *)_IFaceId;
#if 0
  InitData.DriverData.pHWAddr        = "\x00\x22\xC7\xFF\xFF\xF3";
  InitData.DriverData.NumBytesHWAddr = 6;
#endif
  USBD_ECM_Add(&InitData);
}
```

The size of `_abReceiveBuffer` buffer must be a multiple of the USB max packet size. `USB_Driver_IP_NI` is the network interface driver which implements the connection to the IP stack. Optionally a HW address may be configured here, which is assigned to the PC network interface. If not set (`pHWAddr = NULL`), the HW address is generated automatically later while setting the interface up.

The IP stack is configured to use the network interface driver of emUSB-Device-ECM. For more information about the configuration of the IP stack, refer to the *emNet* manual.

 © 2010-2026 SEGGER Microcontroller GmbH

```
// IP_X_Config() - excerpt from IP_Config.c
#include "USB_Driver_IP_NI.h"
void IP_X_Config(void) {
  <...>
  //
  // Add and configure the ECM driver.
  // The local IP address is 10.0.0.10/8.
  //
  IFaceId = IP_AddEtherInterface(&USB_IP_Driver);
  IP_SetAddrMask(0x0A00000A, 0xFF000000);
  IP_SetIFaceConnectHook(IFaceId, _Connect);
  IP_SetIFaceDisconnectHook(IFaceId, _Disconnect);
  _IFaceId = IFaceId;
  <...>
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

15.4 Target API

Function	Description
API functions	
USBD_ECM_Add()	Adds an ECM-class interface to the USB stack.
USBD_ECM_Task()	Obsolete.
Data structures	
USB_ECM_INIT_DATA	Initialization data for ECM interface.
USB_IP_NI_DRIVER_API	This structure contains the callback functions for the network interface driver.
USB_IP_NI_DRIVER_DATA	Configuration data passed to network interface driver at initialization.

 © 2010-2026 SEGGER Microcontroller GmbH

15.4.1 API functions

15.4.1.1 USBD_ECM_Add()

Description

Adds an ECM-class interface to the USB stack.

Prototype

```
void USBD_ECM_Add(const USB_ECM_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a USB_ECM_INIT_DATA structure.

Additional information

This function should be called after the initialization of the USB core to add an ECM interface to emUSB-Device. The initialization data is passed to the function in the structure pointed to by pInitData. Refer to USB_ECM_INIT_DATA for more information.

 © 2010-2026 SEGGER Microcontroller GmbH

15.4.1.2 USBD_ECM_Task()

Description

Obsolete. Returns when USB is disconnected.

Prototype

```
void USBD_ECM_Task(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

15.4.2 Data structures

15.4.2.1 USB_ECM_INIT_DATA

Description

Initialization data for ECM interface.

Type definition

```
typedef struct USB_ECM_INIT_DATA_T {
  U8                             EPIn;
  U8                             EPOut;
  U8                             EPInt;
  const USB_IP_NI_DRIVER_API   * pDriverAPI;
  USB_IP_NI_DRIVER_DATA          DriverData;
  unsigned                       DataInterfaceNo;
} USB_ECM_INIT_DATA;
```

Structure members

Member	Description
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host. The buffer associated to this endpoint must be big enough to hold a complete IP packet.
EPInt	Interrupt IN endpoint for sending status information.
pDriverAPI	Pointer to the Network interface driver API. See USB_IP_NI_DRIVER_API.
DriverData	Configuration data for the network interface driver.
DataInterfaceNo	Internal use.

Additional information

This structure holds the endpoints that should be used by the ECM interface (EPIn, EPOut and EPInt). Refer to USBD_AddEP() for more information about how to add an endpoint.

 © 2010-2026 SEGGER Microcontroller GmbH

15.4.3 Driver interface

15.4.3.1 USB_IP_NI_DRIVER_API

Description

This structure contains the callback functions for the network interface driver.

Type definition

```
typedef struct {
  USB_IP_NI_INIT                      * pfInit;
  USB_IP_NI_GET_PACKET_BUFFER         * pfGetPacketBuffer;
  USB_IP_NI_WRITE_PACKET              * pfWritePacket;
  USB_IP_NI_SET_PACKET_FILTER         * pfSetPacketFilter;
  USB_IP_NI_GET_LINK_STATUS           * pfGetLinkStatus;
  USB_IP_NI_GET_LINK_SPEED            * pfGetLinkSpeed;
  USB_IP_NI_GET_HWADDR                * pfGetHWAddr;
  USB_IP_NI_GET_STATS                 * pfGetStats;
  USB_IP_NI_GET_MTU                   * pfGetMTU;
  USB_IP_NI_RESET                     * pfReset;
  USB_IP_NI_SET_WRITE_PACKET_FUNC     * pfSetWritePacketFunc;
  USB_IP_NI_SET_REPORT_LINKSTATE_FUNC * pfSetReportLinkstateFunc;
} USB_IP_NI_DRIVER_API;
```

Structure members

Member	Description
pfInit	Initializes the driver.
pfGetPacketBuffer	Returns a buffer for a data packet.
pfWritePacket	Delivers a data packet to target IP stack.
pfSetPacketFilter	Configures the type of accepted data packets.
pfGetLinkStatus	Returns the status of the connection to target IP stack.
pfGetLinkSpeed	Returns the connection speed.
pfGetHWAddr	Returns the HW address of the PC.
pfGetStats	Returns statistical counters.
pfGetMTU	Returns the size of the largest data packet which can be transferred.
pfReset	Resets the driver.
pfSetWritePacketFunc	Allows to change the WritePacket callback which was set by pfInit.
pfSetReportLinkstate-Func	Allows to set the report link state change.

Additional information

The emUSB-Device-RNDIS/emUSB-Device-CDC-ECM component calls the functions of this API to exchange data and status information with the IP stack running on the target.

 © 2010-2026 SEGGER Microcontroller GmbH

15.4.3.2 USB_IP_NI_DRIVER_DATA

Description

Configuration data passed to network interface driver at initialization.

Type definition

```
typedef struct USB_IP_NI_DRIVER_DATA {
  const U8   * pHWAddr;
  unsigned     NumBytesHWAddr;
  void       * pDriverData;
  void       * pUSBContext;
} USB_IP_NI_DRIVER_DATA;
```

Structure members

Member	Description
pHWAddr	Optional pointer to a HW address (or MAC address) of the host network interface.
NumBytesHWAddr	Number of bytes in the HW address. Typically 6 bytes.
pDriverData	Pointer to a user context.
pUSBContext	Internal use.

Additional information

When pHWAddr is NULL the MAC is automatically generated.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5 CDC-ECM IP driver

This section describes the emUSB-Device CDC-ECM IP stack interface in detail.

15.5.1 General information

This release comes with an IP NI driver which uses emNet as the IP stack. If you are using emNet, this chapter can be ignored. This chapter is for those who wish to write their own IP stack interface for a third-party IP stack.

The IP interface is handled through an API-table, which contains all relevant functions necessary for read/write operations and initialization.

15.5.2 Interface function list

As described above, access to network functions is realized through an API-function table of type `USB_IP_NI_DRIVER_API`. The structure is declared in `USB_Driver_IP_NI.h`, and it is described in section *Data structures* on page 500.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3 USB_IP_NI_DRIVER_API in detail

15.5.3.1 USB_IP_NI_INIT

Description

Initializes the driver.

Type definition

```
typedef unsigned (USB_IP_NI_INIT)(const USB_IP_NI_DRIVER_DATA * pDriverData,
                                  USB_IP_WRITE_PACKET   * pfWritePacket);
```

Parameters

Parameter	Description
pDriverData	in Pointer to driver configuration data.
pfWritePacket	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

Return value

IP NI driver instance ID.

Additional information

This function is called when the RNDIS/ECM interface is added to the USB stack. Typically the function makes a local copy of the HW address passed in the pDriverData structure. For more information this structure refer to USB_IP_NI_DRIVER_DATA.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.2 USB_IP_NI_GET_PACKET_BUFFER

Description

Returns a buffer for a data packet.

Type definition

```
typedef void * (USB_IP_NI_GET_PACKET_BUFFER)(unsigned Id,
                                             unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
NumBytes	Size of the requested buffer in bytes.

Return value

≠ NULL Pointer to allocated buffer
= NULL No buffer available

Additional information

The function should allocate a buffer of the requested size. If the buffer can not be allocated a NULL pointer should be returned. The function is called when a data packet is received from PC. The packet data is stored in the returned buffer.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.3 USB_IP_NI_WRITE_PACKET

Description

Delivers a data packet to target IP stack.

Type definition

```
typedef void (USB_IP_NI_WRITE_PACKET)(      unsigned Id,
                                      const void   * pData,
                                            unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pData	in Data of the received packet.
NumBytes	Number of bytes stored in the buffer.

Additional information

The function is called after a data packet has been received from USB. pData points to the buffer returned by the USB_IP_NI_GET_PACKET_BUFFER function.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.4 USB_IP_NI_SET_PACKET_FILTER

Description

Configures the type of accepted data packets.

Type definition

```
typedef void (USB_IP_NI_SET_PACKET_FILTER)(unsigned Id,
                                           U32      Mask);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
Mask	Type of accepted data packets.

Additional information

The Mask parameter should be interpreted as a boolean value. A value different than 0 indicates that the connection to target IP stack should be established. When the function is called with the Mask parameter set to 0 the connection to target IP stack should be interrupted.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.5 USB_IP_NI_GET_LINK_STATUS

Description

Returns the status of the connection to target IP stack.

Type definition

```
typedef int (USB_IP_NI_GET_LINK_STATUS)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

Return value

USB_IP_NI_LINK_STATUS_DISCONNECTED	Not connected to target IP stack.
USB_IP_NI_LINK_STATUS_CONNECTED	Connected to target IP stack.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.6 USB_IP_NI_GET_LINK_SPEED

Description

Returns the connection speed.

Type definition

```
typedef U32 (USB_IP_NI_GET_LINK_SPEED)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

Return value

≠ 0	The connection speed in units of 100 bits/sec (between the USB_IP_NI module and the target IP stack).
= 0	Not connected to the target IP stack.

Additional information

In this implementation the return value is 0 when the instance Id was not found or 1000000 (100 Mbit/s).

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.7 USB_IP_NI_GET_HWADDR

Description

Returns the HW address (MAC address) of the host network interface (PC).

Type definition

```
typedef void (USB_IP_NI_GET_HWADDR)(unsigned Id,
                                     U8     * pAddr,
                                     unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pAddr	out The HW address.
NumBytes	Maximum number of bytes to store into pAddr.

Additional information

The returned HW address is the one passed to the driver in the call to USB_IP_NI_INIT. Typically the HW address is 6 bytes long.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.8 USB_IP_NI_GET_STATS

Description

Returns statistical counters.

Type definition

```
typedef U32 (USB_IP_NI_GET_STATS)(unsigned Id,
                                  int      Type);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
Type	The type of information requested. See table below.

Return value

Value of the requested statistical counter.

Additional information

The counters should be set to 0 when the USB_IP_NI_RESET function is called.

Permitted values for parameter Type	
USB_IP_NI_STATS_CLEAR_ALL_STATS	Special type which will instruct the module to reset all statistical counters to zero.
USB_IP_NI_STATS_WRITE_PACKET_OK	Number of packets sent without errors to target IP stack.
USB_IP_NI_STATS_WRITE_PACKET_ERROR	Number of packets sent with errors to target IP stack.
USB_IP_NI_STATS_READ_PACKET_OK	Number of packets received without errors from target IP stack.
USB_IP_NI_STATS_READ_PACKET_ERROR	Number of packets received with errors from target IP stack.
USB_IP_NI_STATS_READ_NO_BUFFER	Number of packets received from target IP stack but dropped.
USB_IP_NI_STATS_READ_ALIGN_ERROR	Number of packets received from target IP stack with alignment errors.
USB_IP_NI_STATS_WRITE_ONE_COLLISION	Number of packets which were not sent to target IP stack due to the occurrence of one collision.
USB_IP_NI_STATS_WRITE_MORE_COLLISIONS	Number of packets which were not sent to target IP stack due to the occurrence of one or more collisions.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.9 USB_IP_NI_GET_MTU

Description

Returns the maximum transmission unit, the size of the largest data packet which can be transferred.

Type definition

```
typedef U32 (USB_IP_NI_GET_MTU)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

Return value

The MTU size in bytes. Typically 1500 bytes.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.10 USB_IP_NI_RESET

Description

Resets the driver.

Type definition

```
typedef void (USB_IP_NI_RESET)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.11 USB_IP_NI_SET_WRITE_PACKET_FUNC

Description

Changes the USB_IP_WRITE_PACKET callback which was added via USB_IP_NI_INIT to a different callback function. This function is only called by the stack when IP-over-USB is used. It is not called when RNDIS or ECM is used standalone.

Type definition

```
typedef void (USB_IP_NI_SET_WRITE_PACKET_FUNC)
                                          (unsigned                Id,
                                           USB_IP_WRITE_PACKET * pfWritePacket);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pfWritePacket	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

 © 2010-2026 SEGGER Microcontroller GmbH

15.5.3.12 USB_IP_NI_SET_REPORT_LINKSTATE_FUNC

Description

Changes the USB_IP_REPORT_LINKSTATE callback. Normally this is called only once per initialization in order to allow to send notification to the host that the link state has been changed.

Type definition

```
typedef void (USB_IP_NI_SET_REPORT_LINKSTATE_FUNC)
                                    (unsigned                   Id,
                                     USB_IP_REPORT_LINKSTATE * pfReportLinkState);
```

Parameters

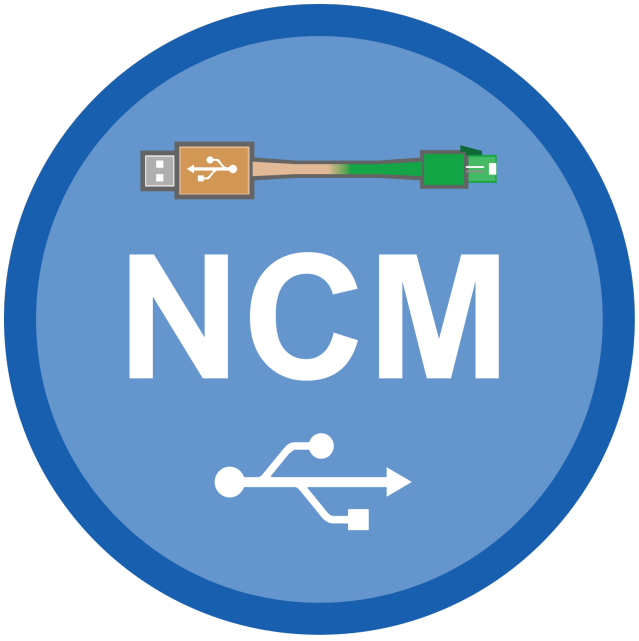
Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pfReportLinkState	Callback function called by the IP stack to notify the host that the link state has been changed.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 16

CDC-NCM

This chapter gives a general overview of the Communications Device Class / Network Control Model class and describes how to get the NCM component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

16.1 Overview

The Communications Device Class / Network Control Model is a USB class protocol of the USB Implementers Forum which can be used to create a virtual Ethernet connection between a USB device and a host PC. A TCP/IP stack like *emNet* is required on the USB device side to handle the actual IP communication. Any available IP protocol (UDP, TCP, FTP, HTTP, etc.) can be used to exchange data.

USB CDC-NCM is supported by Linux (kernel > 2.6.38), macOS, and Windows 10 (starting with build 18362 from May 2019) operating systems out of the box. To use it on old Windows versions, a third-party driver (not contained in emUSB-Device-NCM) has to be installed on the Windows system.

emUSB-Device-NCM contains the following components:

- Generic USB handling
- NCM device class implementation
- Network interface driver which uses *emNet* as a TCP/IP stack.
- Sample application demonstrating how to work with NCM.

16.1.1 Working with CDC-NCM

The `ping` command line utility can be used to test the connection to the target. If the connection is correctly established, the number of the lost packets should be 0.

16.1.2 Additional information

More technical details about CDC-NCM can be found on https://www.usb.org in the Network Control Model Devices Specification v1.0 and errata and Adopters Agreement package.

 © 2010-2026 SEGGER Microcontroller GmbH

16.2 Configuration

16.2.1 Initial configuration

To get emUSB-Device-NCM up and running as well as to do an initial test, the configuration as delivered should not be modified. When using it on old versions of Windows with a third-party driver, the vendor ID and product ID must match the IDs configured in the `.inf` file of the driver.

16.2.2 Final configuration

The configuration must only be modified when emUSB-Device is used in your final product. Refer to section *emUSB-Device Configuration* on page 53 to get detailed information about the general emUSB-Device configuration functions which have to be adapted.

 © 2010-2026 SEGGER Microcontroller GmbH

16.3 Running the sample application

The sample application can be found in the `Sample\NCM\IP_Config_NCM.c` file of the emUSB-Device shipment. In order to use the sample application, the SEGGER *emNet* middleware component is required. To test the emUSB-Device-NCM component, any of the *emNet* sample applications can be used in combination with `IP_Config_NCM.c`. After the sample application is started, the USB cable should be connected to the PC and the chosen *emNet* sample can be tested using the appropriate methods.

16.3.1 IP_Config_NCM.c in detail

The main part of the sample application is implemented in the function `MainTask()` which runs as an independent task.

```
// _Connect() - excerpt from IP_Config_NCM.c
static int _Connect(unsigned IFaceId) {
  U32 Server = IP_BYTES2ADDR(10, 0, 0, 10);
  IP_DHCPS_ConfigPool(IFaceId, IP_BYTES2ADDR(10, 0, 0, 11), 0xFF000000, 20);
  IP_DHCPS_ConfigDNSAddr(IFaceId, &Server, 1);
  IP_DHCPS_Init(IFaceId);
  IP_DHCPS_Start(IFaceId);
  USBD_Init();
  USBD_SetDeviceInfo(&USB_DeviceInfo);
  _AddNCM();
  USBD_Start();
  return 0;                   // Successfully connected.
}
```

The first step is to initialize the DHCP server component which assigns the IP address for the PC side. The target is configured with the IP address 10.0.0.10. The DHCP server is configured to distribute IP addresses starting from 10.0.0.11, therefore the PC will receive the IP address 10.0.0.11. Then the USB stack is initialized and the NCM interface is added to it. The function `_AddNCM()` configures all required endpoints and configures the HW address of the PC network interface.

```
// _AddNCM() - excerpt from IP_Config_NCM.c
static U8 _abReceiveBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
static void _AddNCM(void) {
  USB_NCM_INIT_DATA InitData;
  InitData.EPOut      = USBD_AddEP(USB_DIR_OUT,
                                   USB_TRANSFER_TYPE_BULK,
                                   0,
                                   _abReceiveBuffer, sizeof(_abReceiveBuffer));
  InitData.EPIn       = USBD_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_BULK,
                                   0, NULL, 0);
  InitData.EPInt      = USBD_AddEP(USB_DIR_IN, USB_TRANSFER_TYPE_INT,
                                   32, NULL, 0);
  InitData.pDriverAPI = &USB_Driver_IP_NI;
  InitData.DriverData.pDriverData    = (void *)_IFaceId;
  USBD_NCM_Add(&InitData);
}
```

The size of `_abReceiveBuffer` buffer must be a multiple of USB max packet size. `USB_Driver_IP_NI` is the network interface driver which implements the connection to the IP stack. Optionally a HW address may be configured here, which is assigned to the PC network interface. If not set (`pHWAddr = NULL`), the HW address is generated automatically later while setting the interface up.

The IP stack is configured to use the network interface driver of emUSB-Device-NCM. For more information about the configuration of the IP stack refer to the *emNet* manual.

```
// IP_X_Config() - excerpt from IP_Config.c
#include "USB_Driver_IP_NI.h"
void IP_X_Config(void) {
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
  <...>
  //
  // Add and configure the NCM driver.
  // The local IP address is 10.0.0.10/8.
  //
  IFaceId = IP_AddEtherInterface(&USB_IP_Driver);
IP_SetAddrMask(0x0A00000A, 0xFF000000);
  IP_SetIFaceConnectHook(IFaceId, _Connect);
  IP_SetIFaceDisconnectHook(IFaceId, _Disconnect);
  _IFaceId = IFaceId;
  <...>
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

16.4 Target API

Function	Description
API functions	
USBD_NCM_Add()	Adds an NCM-class interface to the USB stack.
Data structures	
USB_NCM_INIT_DATA	Initialization data for NCM interface.
USB_IP_NI_DRIVER_API	This structure contains the callback functions for the network interface driver.
USB_IP_NI_DRIVER_DATA	Configuration data passed to network interface driver at initialization.

 © 2010-2026 SEGGER Microcontroller GmbH

16.4.1 API functions

16.4.1.1 USBD_NCM_Add()

Description

Adds an NCM-class interface to the USB stack.

Prototype

```
void USBD_NCM_Add(const USB_NCM_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a USB_NCM_INIT_DATA structure.

Additional information

This function should be called after the initialization of the USB core to add an NCM interface to emUSB-Device. The initialization data is passed to the function in the structure pointed to by pInitData. Refer to USB_NCM_INIT_DATA for more information.

 © 2010-2026 SEGGER Microcontroller GmbH

16.4.2 Data structures

16.4.2.1 USB_NCM_INIT_DATA

Description

Initialization data for NCM interface.

Type definition

```
typedef struct USB_NCM_INIT_DATA_T {
  U8                             EPIn;
  U8                             EPOut;
  U8                             EPInt;
  const USB_IP_NI_DRIVER_API   * pDriverAPI;
  USB_IP_NI_DRIVER_DATA          DriverData;
  unsigned                       DataInterfaceNo;
} USB_NCM_INIT_DATA;
```

Structure members

Member	Description
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host. The buffer associated to this endpoint must be big enough to hold a complete IP packet.
EPInt	Interrupt IN endpoint for sending status information.
pDriverAPI	Pointer to the Network interface driver API. See USB_IP_NI_DRIVER_API.
DriverData	Configuration data for the network interface driver.
DataInterfaceNo	Internal use.

Additional information

This structure holds the endpoints that should be used by the NCM interface (EPIn, EPOut and EPInt). Refer to USBD_AddEP() for more information about how to add an endpoint.

 © 2010-2026 SEGGER Microcontroller GmbH

16.4.3 Driver interface

16.4.3.1 USB_IP_NI_DRIVER_API

Description

This structure contains the callback functions for the network interface driver.

Type definition

```
typedef struct {
  USB_IP_NI_INIT                      * pfInit;
  USB_IP_NI_GET_PACKET_BUFFER         * pfGetPacketBuffer;
  USB_IP_NI_WRITE_PACKET              * pfWritePacket;
  USB_IP_NI_SET_PACKET_FILTER         * pfSetPacketFilter;
  USB_IP_NI_GET_LINK_STATUS           * pfGetLinkStatus;
  USB_IP_NI_GET_LINK_SPEED            * pfGetLinkSpeed;
  USB_IP_NI_GET_HWADDR                * pfGetHWAddr;
  USB_IP_NI_GET_STATS                 * pfGetStats;
  USB_IP_NI_GET_MTU                   * pfGetMTU;
  USB_IP_NI_RESET                     * pfReset;
  USB_IP_NI_SET_WRITE_PACKET_FUNC     * pfSetWritePacketFunc;
  USB_IP_NI_SET_REPORT_LINKSTATE_FUNC * pfSetReportLinkstateFunc;
} USB_IP_NI_DRIVER_API;
```

Structure members

Member	Description
pfInit	Initializes the driver.
pfGetPacketBuffer	Returns a buffer for a data packet.
pfWritePacket	Delivers a data packet to target IP stack.
pfSetPacketFilter	Configures the type of accepted data packets.
pfGetLinkStatus	Returns the status of the connection to target IP stack.
pfGetLinkSpeed	Returns the connection speed.
pfGetHWAddr	Returns the HW address of the PC.
pfGetStats	Returns statistical counters.
pfGetMTU	Returns the size of the largest data packet which can be transferred.
pfReset	Resets the driver.
pfSetWritePacketFunc	Allows to change the WritePacket callback which was set by pfInit.
pfSetReportLinkstate-Func	Allows to set the report link state change.

Additional information

The emUSB-Device-RNDIS/emUSB-Device-CDC-ECM component calls the functions of this API to exchange data and status information with the IP stack running on the target.

 © 2010-2026 SEGGER Microcontroller GmbH

16.4.3.2 USB_IP_NI_DRIVER_DATA

Description

Configuration data passed to network interface driver at initialization.

Type definition

```
typedef struct USB_IP_NI_DRIVER_DATA {
  const U8   * pHWAddr;
  unsigned     NumBytesHWAddr;
  void       * pDriverData;
  void       * pUSBContext;
} USB_IP_NI_DRIVER_DATA;
```

Structure members

Member	Description
pHWAddr	Optional pointer to a HW address (or MAC address) of the host network interface.
NumBytesHWAddr	Number of bytes in the HW address. Typically 6 bytes.
pDriverData	Pointer to a user context.
pUSBContext	Internal use.

Additional information

When pHWAddr is NULL the MAC is automatically generated.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5 CDC-NCM IP Driver

This section describes the emUSB-Device CDC-NCM IP stack interface in detail.

16.5.1 General information

This release comes with an IP NI driver which uses emNet as the IP stack. If you are using emNet, this chapter can be ignored. This chapter is for those who wish to write their own IP stack interface for a third-party IP stack.

The IP interface is handled through an API-table, which contains all relevant functions necessary for read/write operations and initialization.

16.5.2 Interface function list

As described above, access to network functions is realized through an API-function table of type `USB_IP_NI_DRIVER_API`. The structure is declared in `USB_Driver_IP_NI.h`, and it is described in section *Data structures* on page 500.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3 USB_IP_NI_DRIVER_API in detail

16.5.3.1 USB_IP_NI_INIT

Description

Initializes the driver.

Type definition

```
typedef unsigned (USB_IP_NI_INIT)(const USB_IP_NI_DRIVER_DATA * pDriverData,
                                  USB_IP_WRITE_PACKET   * pfWritePacket);
```

Parameters

Parameter	Description
pDriverData	in Pointer to driver configuration data.
pfWritePacket	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

Return value

IP NI driver instance ID.

Additional information

This function is called when the RNDIS/ECM interface is added to the USB stack. Typically the function makes a local copy of the HW address passed in the pDriverData structure. For more information this structure refer to USB_IP_NI_DRIVER_DATA.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.2 USB_IP_NI_GET_PACKET_BUFFER

Description

Returns a buffer for a data packet.

Type definition

```
typedef void * (USB_IP_NI_GET_PACKET_BUFFER)(unsigned Id,
                                             unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
NumBytes	Size of the requested buffer in bytes.

Return value

≠ NULL Pointer to allocated buffer
= NULL No buffer available

Additional information

The function should allocate a buffer of the requested size. If the buffer can not be allocated a NULL pointer should be returned. The function is called when a data packet is received from PC. The packet data is stored in the returned buffer.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.3 USB_IP_NI_WRITE_PACKET

Description

Delivers a data packet to target IP stack.

Type definition

```
typedef void (USB_IP_NI_WRITE_PACKET)(      unsigned Id,
                                      const void   * pData,
                                            unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pData	in Data of the received packet.
NumBytes	Number of bytes stored in the buffer.

Additional information

The function is called after a data packet has been received from USB. pData points to the buffer returned by the USB_IP_NI_GET_PACKET_BUFFER function.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.4 USB_IP_NI_SET_PACKET_FILTER

Description

Configures the type of accepted data packets.

Type definition

```
typedef void (USB_IP_NI_SET_PACKET_FILTER)(unsigned Id,
                                           U32      Mask);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
Mask	Type of accepted data packets.

Additional information

The Mask parameter should be interpreted as a boolean value. A value different than 0 indicates that the connection to target IP stack should be established. When the function is called with the Mask parameter set to 0 the connection to target IP stack should be interrupted.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.5 USB_IP_NI_GET_LINK_STATUS

Description

Returns the status of the connection to target IP stack.

Type definition

```
typedef int (USB_IP_NI_GET_LINK_STATUS)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

Return value

USB_IP_NI_LINK_STATUS_DISCONNECTED	Not connected to target IP stack.
USB_IP_NI_LINK_STATUS_CONNECTED	Connected to target IP stack.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.6 USB_IP_NI_GET_LINK_SPEED

Description

Returns the connection speed.

Type definition

```
typedef U32 (USB_IP_NI_GET_LINK_SPEED)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

Return value

≠ 0	The connection speed in units of 100 bits/sec (between the USB_IP_NI module and the target IP stack).
= 0	Not connected to the target IP stack.

Additional information

In this implementation the return value is 0 when the instance Id was not found or 1000000 (100 Mbit/s).

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.7 USB_IP_NI_GET_HWADDR

Description

Returns the HW address (MAC address) of the host network interface (PC).

Type definition

```
typedef void (USB_IP_NI_GET_HWADDR)(unsigned Id,
                                     U8     * pAddr,
                                     unsigned NumBytes);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pAddr	out The HW address.
NumBytes	Maximum number of bytes to store into pAddr.

Additional information

The returned HW address is the one passed to the driver in the call to USB_IP_NI_INIT. Typically the HW address is 6 bytes long.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.8 USB_IP_NI_GET_STATS

Description

Returns statistical counters.

Type definition

```
typedef U32 (USB_IP_NI_GET_STATS)(unsigned Id,
                                  int      Type);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
Type	The type of information requested. See table below.

Return value

Value of the requested statistical counter.

Additional information

The counters should be set to 0 when the USB_IP_NI_RESET function is called.

Permitted values for parameter Type	
USB_IP_NI_STATS_CLEAR_ALL_STATS	Special type which will instruct the module to reset all statistical counters to zero.
USB_IP_NI_STATS_WRITE_PACKET_OK	Number of packets sent without errors to target IP stack.
USB_IP_NI_STATS_WRITE_PACKET_ERROR	Number of packets sent with errors to target IP stack.
USB_IP_NI_STATS_READ_PACKET_OK	Number of packets received without errors from target IP stack.
USB_IP_NI_STATS_READ_PACKET_ERROR	Number of packets received with errors from target IP stack.
USB_IP_NI_STATS_READ_NO_BUFFER	Number of packets received from target IP stack but dropped.
USB_IP_NI_STATS_READ_ALIGN_ERROR	Number of packets received from target IP stack with alignment errors.
USB_IP_NI_STATS_WRITE_ONE_COLLISION	Number of packets which were not sent to target IP stack due to the occurrence of one collision.
USB_IP_NI_STATS_WRITE_MORE_COLLISIONS	Number of packets which were not sent to target IP stack due to the occurrence of one or more collisions.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.9 USB_IP_NI_GET_MTU

Description

Returns the maximum transmission unit, the size of the largest data packet which can be transferred.

Type definition

```
typedef U32 (USB_IP_NI_GET_MTU)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

Return value

The MTU size in bytes. Typically 1500 bytes.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.10 USB_IP_NI_RESET

Description

Resets the driver.

Type definition

```
typedef void (USB_IP_NI_RESET)(unsigned Id);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.11 USB_IP_NI_SET_WRITE_PACKET_FUNC

Description

Changes the USB_IP_WRITE_PACKET callback which was added via USB_IP_NI_INIT to a different callback function. This function is only called by the stack when IP-over-USB is used. It is not called when RNDIS or ECM is used standalone.

Type definition

```
typedef void (USB_IP_NI_SET_WRITE_PACKET_FUNC)
                                          (unsigned              Id,
                                           USB_IP_WRITE_PACKET * pfWritePacket);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pfWritePacket	Callback function called by the IP stack to transmit a packet that should be sent to the USB host.

 © 2010-2026 SEGGER Microcontroller GmbH

16.5.3.12 USB_IP_NI_SET_REPORT_LINKSTATE_FUNC

Description

Changes the USB_IP_REPORT_LINKSTATE callback. Normally this is called only once per initialization in order to allow to send notification to the host that the link state has been changed.

Type definition

```
typedef void (USB_IP_NI_SET_REPORT_LINKSTATE_FUNC)
                                    (unsigned                   Id,
                                     USB_IP_REPORT_LINKSTATE * pfReportLinkState);
```

Parameters

Parameter	Description
Id	Instance ID returned from USB_IP_NI_INIT.
pfReportLinkState	Callback function called by the IP stack to notify the host that the link state has been changed.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 17

Audio

This chapter gives a general overview of the Audio class and describes how to get the Audio component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

17.1 Overview

The USB Audio device class is a USB class protocol which can be used to transfer sound data from a device to a host and vice versa.

Audio is supported by most operating systems out of the box and the installation of additional drivers is not required.

emUSB-Device Audio provides the following features:

- Compatible with USB Audio version 1 and version 2 device class.
- Arbitrary number of input and output audio streams.
- Arbitrary number of channels for each audio stream.
- Sample resolution from 8- to 32-bit.
- Arbitrary number of audio control units.
- Synchronous, asynchronous, adaptive and implicit synchronization for each audio stream.
- Audio interrupt messages.

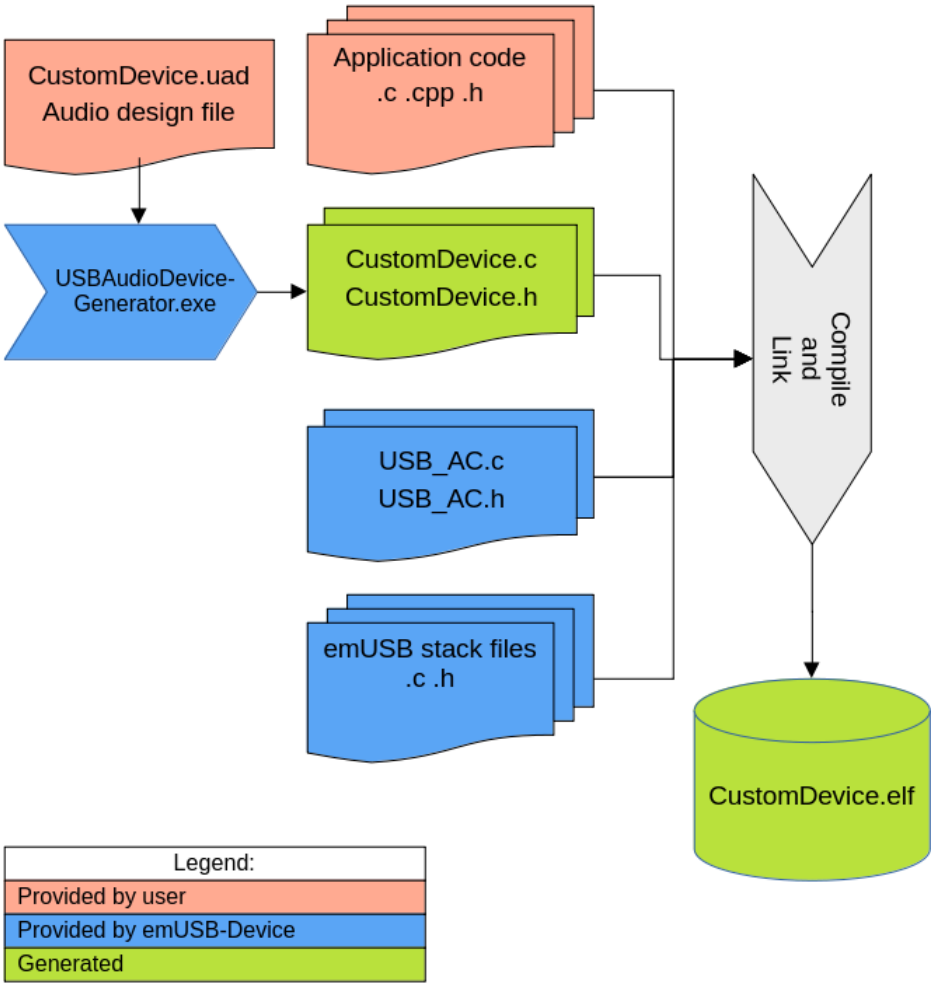
The emUSB-Device implementation of the Audio class is designed with minimal resource usage in mind, especially targeted to embedded devices. emUSB-Device-Audio supports the transparent transport of audio data to and from a USB host, but does not care about the format of the audio data (number of channels, bit resolution, encoding). Generation and processing of correctly formatted audio data is up to the application. The application also has to consider how the stream of audio samples must be split into USB packets.

> **Note**
>
> Audio 2.0 is designed for high-speed USB devices and should not be used on full-speed devices. It may work on full-speed devices with modified configurations, but there is no official support in emUSB-Device.

 © 2010-2026 SEGGER Microcontroller GmbH

17.2 Creation of an audio device application

A USB audio device is a collection of audio control units, audio streaming interfaces and alternate interface settings. To build an audio device, the design of this device must be defined by creating a "USB audio design" file (extension `.uad`). This file specifies all characteristics of the device and is converted by the `USBAudioDeviceGenerator.exe` tool into a C source file and a C header file that should be used to build the audio application.



emUSB-Device contains a couple of sample USB audio design files and sample application code that can be used as a starting point to develop your own audio application.

The USB Audio Device Generator tool is a command-line tool that can be invoked from a command shell:

```
USBAudioDeviceGenerator.exe [-s] [-o=<output-file>] <USB-audio-design-file>

  -s    Silent execution (except for errors).
  -o=   Base name for the generated .c and .h files.
        If not specified, the name of the USB audio design file is used.
```

To register an audio class at the USB stack, the application has to create a `USBD_AC_INIT_DATA` structure and call the function `USBD_AC_Add()`. The structure has to contain a pointer to the configuration data created by the generator and three callback functions that must be provided by the application: The function `pfSetAlternate` is called every time the host selects a new alternate setting for any of the audio interfaces. The other two functions must handle audio control get/set requests sent by the host.

 © 2010-2026 SEGGER Microcontroller GmbH

For audio 2.0 devices, it is required to call `USBD_EnableIAD()` before `USBD_AC_Add()`.

17.2.1 Configuration requirements

In order to use the audio class, isochronous transfer support must be enabled in the USB stack by setting *USB_SUPPORT_TRANSFER_ISO* to 1. Additionally, it may be necessary to increase the values of the following compile-time configuration options:

- *USB_MAX_NUM_IF*
- *USB_MAX_NUM_ALT_IF*
- *USB_DESC_BUFFER_SIZE*

17.2.2 Design of audio interfaces

An audio 1.0 interface usually has multiple alternate settings, one for each supported sample frequency. Alternate setting 0 does not contain any endpoints and is selected by the host to switch off the interface. Selecting an alternate setting by the host (other than 0) implies selecting a particular sample frequency.

An audio 2.0 interface usually has only alternate settings '0' and '1', which are used by the host to switch the interface on and off. Different sample frequencies are handled using audio control requests sent to a clock unit. Audio 2.0 interfaces may have more alternate settings as well to provide a variable bit resolution or a variable number of channels.

While audio 1.0 devices transfer one packet of audio data every millisecond, audio 2.0 devices may transfer 1, 2, 4 or 8 packets every millisecond (one packet every 1ms, 500µs, 250µs, 125µs respectively). Choosing a smaller interval for an interface leads to smaller packet sizes but may require a more strict response time by the application to handle data packets.

For each alternate setting of an interface containing an endpoint, a maximum packet size (in bytes) must be configured. It must be large enough to hold all audio samples for the given interval.

Example

```
16-bit stereo               --> Sample size 4 bytes
Sample frequency 48000 Hz
Interval 1ms                --> 48 samples per packet
                            --> maximum packet size: 4 * 48 = 192 bytes
```

For configurations where not all packets contain the same number of audio samples, the maximum packet size needs to be rounded up.

Example

```
16-bit stereo               --> Sample size 4 bytes
Sample frequency 44100 Hz
Interval 1ms                --> packets contain 44 or 45 samples
                            --> maximum packet size: 4 * 45 = 180 bytes
```

If explicit synchronization is used (asynchronous OUT or adaptive IN) the packet must be able to hold at least one more audio sample to allow proper feedback response.

Example

```
24-bit stereo               --> Sample size 6 bytes
Sample frequency 96000 Hz
Interval 250us              --> 24 samples per packet
Asynchronous OUT            --> maximum packet size: 6 * (24+1) = 150 bytes
```

Warning

 © 2010-2026 SEGGER Microcontroller GmbH

During enumeration of a device all alternate settings with endpoint configurations that are not compatible with the enumerated speed are dropped from the configuration descriptor. This will happen for example for alternate settings with an endpoint size > 1023 or an interval < 1ms when enumerated in full-speed. As a result, the audio device may not be usable when enumerated in a lower speed.

17.2.3 Handling of audio control requests

If an audio control request is received from the host, one of the callback functions registered with the USBD_AC_INIT_DATA structure is called. In these functions the application must handle all controls that were configured in the USB audio description file with any "Controls:" statement.

17.2.4 Receiving audio data

If the host wants to send audio data to the device, it will select an appropriate alternate setting on an OUT interface. The application is notified via the pfSetAlternate callback function. In order to receive audio data from the host, the application then has to create and initialize a data structure of type USBD_AC_RX_CTX. This contains a pointer to a function of the application (pfCallback) that is called by the audio class every time audio data is received from the host. Additionally, the USBD_AC_RX_DATA structure (contained in USBD_AC_RX_CTX) must be initialized: The application has to provide a buffer where the audio data is stored and can configure the audio class to read a single or multiple packets into this buffer before the callback function is called.

Then the application has to call USBD_AC_OpenRXStream() which starts reading data. The contents of the USBD_AC_RX_CTX structure (memory area) must be valid and must not be modified while the stream is open.

Every time the requested amount of audio data is received, the callback function of the application (pfCallback) is called. The received audio data must not be processed within this function, because it is called from an interrupt context. Instead, the function must reinitialize the USBD_AC_RX_DATA structure (to enable reading of the next data) and return immediately. Audio data should be processed in an application task; therefore, at least double buffering is recommended: The callback function should modify the USBD_AC_RX_DATA structure to use a different buffer to not overwrite the data received before while it is processed by the application.

17.2.4.1 Using explicit feedback

If explicit feedback is used for a RX audio stream, the application has to compute periodically the number of audio samples that are processed within the data transfer interval defined by the USB host controller and provide this information to the host using the function USBD_AC_SetFeedbackDataRate(). In order to synchronize with the USB clock of the host the application may register a callback function that is called in regular intervals (related to the USB clock) using the members pfSOFCallback and FeedbackInterval of the USBD_AC_RX_CTX structure. In this function the application can measure the number of samples processed (using the clock source of the audio hardware) and calculate the feedback value.

17.2.5 Sending audio data

If the host wants to read audio data from the device, it will select an appropriate alternate setting on an IN interface. The application is notified via the pfSetAlternate callback function. In order to send audio data to the host, the application then has to create and initialize a data structure of type USBD_AC_TX_CTX and call the function USBD_AC_OpenTXStream(). The contents of the USBD_AC_TX_CTX structure (memory area) must be valid and must not be modified while the stream is open.

After successful open, the application can start to send audio data with USBD_AC_Send(). This function returns immediately, performing the data transfer asynchronously. A maxi-

 © 2010-2026 SEGGER Microcontroller GmbH

mum of two calls to USBD_AC_Send() can be queued by the audio class. Every time the data from a USBD_AC_Send() was transferred to the host, a callback function provided by the application (member pfCallback in USBD_AC_TX_CTX) is executed in order to signal that new audio data can be accepted to be sent. This function must not generate new audio data itself, because it is called from an interrupt context. Instead, it may trigger an application task to create more audio data and call another USBD_AC_Send().

The send queue is used to support a double buffering of audio data: While a data packet is sent to the host, another data packet can be created by the application and queued with USBD_AC_Send().

17.2.5.1 Using explicit feedback

If explicit feedback is used for a TX audio stream, the application has to fetch the feedback information from the host using USBD_AC_GetFeedbackDataRate() periodically and adjust the number of audio samples that are sent accordingly.

17.2.6 Physical controls

If the audio device has any kind of physical controls, like a volume control button, there are two different ways to handle these buttons by the application:

Physical button is a HID control

In this case, the physical button is completely separate from the audio function and is implemented within a separate HID interface. The audio function is not even aware of the button's existence. Any change of state for the button is communicated to the host software via HID reports. It is then up to the host software to interpret the button state change and send an appropriate control request to the audio function.

Physical button is an integral part of the audio control

In this case, the physical button directly interacts with the actual audio control, causing the state of an audio control unit to change directly. The application should inform the host through the audio control interrupt mechanism by calling the function USBD_AC_SendInterruptMessage(). The host then can retrieve the current setting of the audio control unit using a control get request.

In order to use the audio control interrupt mechanism, an interrupt endpoint must be configured in the audio control interface, see member IntEP in USBD_AC_INIT_DATA.

 © 2010-2026 SEGGER Microcontroller GmbH

17.3 Syntax definition of the USB audio design file

The USB audio design file is an ASCII file containing the design description of a USB audio device given in a syntax described within this section. To create such a file, knowledge of concepts and terms of USB audio devices is required, see the document "Universal Serial Bus Device Class Definition for Audio Devices" from www.usb.org.

The file can contain C++-like comments at any location which start with // and include the rest of the current line. Any number of spaces or newlines can be present between keywords. The following terms are used in the syntax definition in this section:

```
'<ID>'          := Identifier. <ID> must begin with a letter and only consist
                   of letters (A-Z, a-z), digits (0-9) or underscore characters

"<string>"      := Arbitrary text description containing a maximum
                   of 255 characters

0int            := An integer number, either in decimal or
                   in hexadecimal (preceded with "0x")

#placeholder#   := Placeholder for an element described below in the section.

#item# ...      := A list of one or more #item# elements

#item# ,...     := A comma-separated list of one or more #item# elements
```

An identifier uniquely identifies a control unit or streaming interface and can be used to define connections between these entities. When used before a { it is defined and assigned to the current unit. It can then be used as a reference in other units. Example:

```
CLOCK_SOURCE 'Clock1' {      // Gives this unit the name "Clock1"
  ...
}

INPUT_TERMINAL 'Term1' {     // Gives this unit the name "Term1"
  ...
  ClockSource: 'Clock1';     // References the Clock Source unit above
                             // as clock input
  ...
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

17.3.1 Overall syntax of the design file

```
AudioDevice '<ID>' #version# {

  Category:     #category#;
  Description:  "<string>";
  Controls:     #MainControl# ,... ;

  #ControlUnit# ...

  #Interface#   ...

  #CompilerMacros#
}
```

The description must contain at least one `#ControlUnit#` and one `#Interface#`; other elements are optional.

`#ControlUnit#` is one of the elements described in the section *Control units description*.

`#Interface#` is an interface description defined in the section *Streaming interface description*.

```
#version# is either 1.0 or 2.0

#category# is one of the following:
DESKTOP_SPEAKER
HOME_THEATER
MICROPHONE
HEADSET
TELEPHONE
CONVERTER
VOICE/SOUND_RECORDER
I/O_BOX
MUSICAL_INSTRUMENT
PRO-AUDIO
AUDIO/VIDEO
CONTROL_PANEL
OTHER

#MainControl# is one of the following:
LatencyControl
```

17.3.1.1 Compiler Macros

The `#CompilerMacros#` statement can be used to create user-defined preprocessor defines stored into the generated header file. These have no effect on the generated audio configuration and can be arbitrarily used by the audio application.

Example

```
CompilerMacros {
  'XXX_VALUE'     "4711";
  'XXX_SIZE'      "(sizeof(XXX) + 3)";
  'XXX_NAME'      "\''Mega_XXX\''";
}
```

will create the following defines in the header file:

```
#define XXX_VALUE     4711
#define XXX_SIZE      (sizeof(XXX) + 3)
#define XXX_NAME      "Mega_XXX"
```

 © 2010-2026 SEGGER Microcontroller GmbH

17.3.2 Control units description

`#ControlUnit#` is one of the elements described below.

17.3.2.1 Input Terminal

```
INPUT_TERMINAL '<ID>' {
  TerminalType:   0int;
  ClockSource:    '<ID>';
  ChannelCluster: #ChannelLocation# ,... ;
  Controls:       #InputTerminalControl# ,... ;
  AssocTerminal:  '<ID>';
  Description:    "<string>";
}
```

All items of this unit are optional, except `ChannelCluster` and `ClockSource` (for audio version 2.0 devices only). Valid numbers for `TerminalType` can be found in the document "Universal Serial Bus Device Class Definition for Terminal Types" from www.usb.org. `[{Controls}]` is valid for audio 2.0 devices only.

```
#ChannelLocation# is one of the following:

FrontLeft
LeftFront
FrontRight
RightFront
FrontCenter
CenterFront
LowFrequencyEffects
BackLeft
LeftSurround
BackRight
RightSurround
FrontLeftOfCenter
LeftOfCenter
FrontRightOfCenter
RightOfCenter
BackCenter
Surround
SideLeft
SideRight
TopCenter
Top
TopFrontLeft
TopFrontCenter
TopFrontRight
TopBackLeft
TopBackCenter
TopBackRight
TopFrontLeftOfCenter
TopFrontRightOfCenter
LeftLowFrequencyEffects
RightLowFrequencyEffects
TopSideLeft
TopSideRight
BottomCenter
BackLeftOfCenter
BackRightOfCenter
RawData
NoLocation       // Unspecified non-standard Channel Cluster location
"<string>"       // Non-standard Channel Cluster location described by <string>
```

```
#InputTerminalControl# is one of the following:

OverloadControl
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
ConnectorControl
CopyProtectControl
UnderflowControl
OverflowControl
ClusterControl
//
// All of these may optionally be followed by the "ReadOnly" keyword
//
```

17.3.2.2 Output Terminal

```
OUTPUT_TERMINAL '<ID>' {
  TerminalType:   0int;
  ClockSource:    '<ID>';
  Input:          '<ID>' ,... ;
  Controls:       #OutputTerminalControl# ,... ;
  AssocTerminal:  '<ID>';
  Description:    "<string>";
}
```

All items of this unit are optional, except `Input` and `ClockSource` (for audio version 2.0 devices only). Valid numbers for `TerminalType` can be found in the document “Universal Serial Bus Device Class Definition for Terminal Types” from www.usb.org. `[{Controls}]` is valid for audio 2.0 devices only.

```
#OutputTerminalControl# is one of the following:

OverloadControl
ConnectorControl
CopyProtectControl
UnderflowControl
OverflowControl
//
// All of these may optionally be followed by the "ReadOnly" keyword
//
```

17.3.2.3 Feature unit

```
FEATURE_UNIT '<ID>' {
  Input:          '<ID>' ,... ;
  Controls:       #FeatureControl# ,... ;
  Description:    "<string>";
}
```

All items of this unit are optional, except `Input`.

```
#FeatureControl# is one of the following:

MuteControl
VolumeControl
BassControl
MidControl
TrebleControl
GraphicEqualizerControl
AutomaticGainControl
DelayControl
BassBoostControl
LoudnessControl
InputGainControl          // Audio 2.0 only
InputGainPadControl       // Audio 2.0 only
PhaseInverterControl      // Audio 2.0 only
UnderflowControl          // Audio 2.0 only
OverflowControl           // Audio 2.0 only
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
//
// For audio version 2.0 devices all of these may
// optionally be followed by the "ReadOnly" keyword
//
```

17.3.2.4 Mixer unit

```
MIXER_UNIT '<ID>' {
  Input:          '<ID>' ,... ;
  Controls:       #MixerControl# ,... ;
  ChannelCluster: #ChannelLocation# ,... ;
  MixerControls:  #MixerMatrix#;
  Description:    "<string>";
}
```

All items of this unit are required, except `Controls` and `Description`.

```
#MixerControl# is one of the following:

ClusterControl
UnderflowControl
OverflowControl
//
// For audio version 2.0 devices all of these may
// optionally followed by the "ReadOnly" keyword
//
```

`#MixerMatrix#` is a two-dimensional bit array that has a row for each logical input channel and a column for each logical output channel. If a bit at position [x,y] is set, this means that the Mixer Unit contains a programmable mixing Control that connects input channel x to output channel y. If bit [x,y] is clear, this indicates that the connection between input channel x and output channel y is non-programmable. The number of input channels is defined by `Input`, and the number of output channels is specified by `ChannelCluster`.

`#MixerMatrix#` is a comma-separated list of rows. Each row is a list of bits (’0’ and ’1’ characters) enclosed in square brackets. Example:

```
MIXER_UNIT 'Mixer' {
  Input: 'LineIN', 'MicIN';
  ChannelCluster: FrontLeft, FrontRight, "Floor Vibration";
  //                                                Input channels (rows):
  MixerControls: [ 1          1           1               ],  // Line Left
                 [ 1          1           1               ],  // Line Right
                 [ 1          1           0               ],  // Mic Left
                 [ 1          1           0               ];  // Mic Right
  // Out channels: FrontLeft  FrontRight  Floor Vibration
  Description: "Line / Mic mixer";
}
```

17.3.2.5 Selector unit

```
SELECTOR_UNIT '<ID>' {
  Input:          '<ID>' ,... ;
  Controls:       #SelectorControl# ,... ;
  Description:    "<string>";
}
```

All items of this unit are optional, except `Input`. `Controls` is valid for audio 2.0 devices only.

```
#SelectorControl# is one of the following:

SelectorControl
//
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
// All of these may optionally followed by the "ReadOnly" keyword
//
```

17.3.2.6 Clock source

This unit is allowed for audio 2.0 devices only.

```
CLOCK_SOURCE '<ID>' {
  Type:           #ClockType# ;
  AssocTerminal:  '<ID>';
  Controls:       #ClockSourceControl# ,... ;
  Description:    "<string>";
}
```

All items of this unit are optional.

```
#ClockType# is one of the following:

External
Internal fixed
Internal fixed SynchonizedToSOF
Internal variable
Internal variable SynchonizedToSOF
Internal programmable
Internal programmable SynchonizedToSOF
```

```
#ClockSourceControl# is one of the following:

ClockFrequencyControl
ClockValidityControl
//
// All of these may optionally be followed by the "ReadOnly" keyword
//
```

17.3.2.7 Clock selector

This unit is allowed for audio 2.0 devices only.

```
CLOCK_SELECTOR '<ID>' {
  Input:          '<ID>' ,... ;
  Controls:       #ClockSelectorControl# ,... ;
  Description:    "<string>";
}
```

All items of this unit are optional, except `Input`.

```
#ClockSelectorControl# is one of the following:

ClockSelectorControl
//
// All of these may optionally followed by the "ReadOnly" keyword
//
```

17.3.2.8 Clock multiplier

This unit is allowed for audio 2.0 devices only.

```
CLOCK_MULTIPLIER '<ID>' {
  Input:          '<ID>' ,... ;
  Controls:       #ClockMultiplierControl# ,... ;
  Description:    "<string>";
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

All items of this unit are optional, except `Input`.

```
#ClockMultiplierControl# is one of the following:

ClockNumeratorControl
ClockDenominatorControl
//
// All of these may optionally followed by the "ReadOnly" keyword
//
```

 © 2010-2026 SEGGER Microcontroller GmbH

17.3.3 Streaming interface description

The overall syntax of an interface description #Interface# is:

```
StreamingInterface '<ID>' {
  #AlternateConfig# ...
  #CompilerMacros#
}
```

#CompilerMacros# are defined in *Compiler Macros* on page 583.

#AlternateConfig# is defined as:

```
Alternate {
  AUDIO_STREAM {
    Description:      "<string>";
    Terminal:         '<ID>';           // Reference to an Input- or
                                        // Output Terminal control unit
    //
    // The following items are valid for USB audio 1.0 devices only
    //
    FormatTag:        0int;
    Delay:            0int;
    //
    // The following items are valid for USB audio 2.0 devices only
    //
    ChannelCluster:   #ChannelLocation# ,... ;
    FormatType:       #FormatType#;
    Formats:          0int;
    Controls:         #StreamControl# ,... ;
  }

  ENDPOINT {
    Direction:        #Direction#;
    MaxPacketSize:    0int;
    Interval:         0int;
    FeedbackInterval: 0int;
    Sync:             #Synchronisation#;
    Attributes:       #EndpointAttribute# ,... ;
    Controls:         #EndpointControl# ,... ;
    LockDelay:        0int 0int;
  }

  #StreamUnit# ...
}
```

The AUDIO_STREAM section is mandatory and must contain at least the Terminal and ChannelCluster entries.

The ENDPOINT section is optional but must contain at least the Direction and MaxPacketSize entries if present.

#StreamUnit# is one of the elements described in the section *Stream units description*.

17.3.3.1 AUDIO_STREAM section

FormatTag describes the Audio Data Format that should be used when exchanging data with this endpoint. A complete list of supported Audio Data Formats is provided in the document "Universal Serial Bus Device Class Definition for Audio Data Formats 1.0" from www.usb.org.

Delay specifies the delay (δ) introduced by the data path expressed in number of frames.

#ChannelLocation# is defined in section *Input Terminal*.

```
#FormatType# is one of the following:
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
1
2
3
4
Extended 1
Extended 2
Extended 3
```

`Formats` specifies an integer containing a bitmap that lists the Audio Data Format(s) that can be used to communicate with this interface. See the document "Universal Serial Bus Device Class Definition for Audio Data Formats 2.0" from www.usb.org for further details.

```
#StreamControl# is one of the following:

ActiveAlternateSetting
ValidAlternateSetting
```

17.3.3.2 ENDPOINT section

`#Direction#` is either `IN` (data transfer from the device to the host) or `OUT` (data transfer from the host to the device).

`MaxPacketSize` specifies the maximum number of bytes transferred within one packet. Up to 1023 bytes are allowed for full-speed devices and up to 3072 bytes for high-speed devices. More information about the maximum packet size can be found in section *Design of audio interfaces* on page 579.

`Interval` defines the time between sending two audio packets. For audio 1.0 devices it must be given in milliseconds (default is 1ms). For audio 2.0 devices it must be specified in units of 125µs. A value of 1, 2, 4 or 8 defines an interval of 125µs, 250µs, 500µs or 1ms respectively. The value must always be a power of 2.

`FeedbackInterval` defines the interval the feedback endpoint should be polled. It must be specified only if a feedback endpoint is used for this interface.

```
#Synchronisation# is one of the following:

Synchronous
Asynchronous
Asynchronous Implicit
Adaptive
Adaptive Implicit


#EndpointAttribute# is one of the following:

SamplingFrequencyControl    // audio 1.0 only
PitchControl                // audio 1.0 only
MaxPacketsOnly


#EndpointControl# is valid for audio 2.0 only and is one of the following:

PitchControl
DataOverrunControl
DataUnderrunControl
//
// All of these may optionally be followed by the "ReadOnly" keyword
//
```

`LockDelay` specifies values for the `bLockDelayUnits` and `wLockDelay` parameters. These are used to indicate to the Host how long it takes for the clock recovery circuitry of this endpoint to lock and reliably produce or consume the audio data stream.

 © 2010-2026 SEGGER Microcontroller GmbH

17.3.4 Stream units description

`#StreamUnit#` is one of the elements described below.

17.3.4.1 Format I section

```
FORMAT_I {
  SubframeSize:           0int;     // Audio 1.0
  SubslotSize:            0int;     // Audio 2.0
  BitResolution:          0int;
  //
  // The following items are valid for USB audio 1.0 devices only
  //
  NrChannels:             0int;
  SamplingFrequency:      0int ...;
  SamplingFrequencyRange: 0intMin 0intMax;
}
```

`SubframeSize` / `SubslotSize` defines the number of bytes occupied by one audio sample for a single channel. The value can be 1, 2, 3, or 4.

`BitResolution` specifies the number of effectively used bits from the available bits in an audio subframe / subslot. The value must be ≤ 8 * `SubframeSize` or `SubslotSize`.

`NrChannels` indicates the number of physical channels in the audio data stream.

Supported sample frequencies can be specified either with a list of discrete frequencies (`SamplingFrequency`) or by a frequency range with minimum and maximum values (`SamplingFrequencyRange`). All frequency values must be given in Hz.

17.3.4.2 Format II section

This format is used to transfer encoded audio data.

```
FORMAT_II {
  MaxBitRate:             0int;
  SamplesPerFrame:        0int;     // Audio 1.0
  SlotsPerFrame:          0int;     // Audio 2.0
  //
  // The following items are valid for USB audio 1.0 devices only
  //
  SamplingFrequency:      0int ...;
  SamplingFrequencyRange: 0intMin 0intMax;
}
```

`MaxBitRate` specifies the maximum number of bits per second this interface can handle. It is a measure for the buffer size available in the interface.

`SamplesPerFrame` indicates the number of PCM audio samples contained in one encoded audio frame.

`SlotsPerFrame` contains the number of PCM audio slots contained within a single encoded audio frame.

Supported sample frequencies can be specified either with a list of discrete frequencies (`SamplingFrequency`) or by a frequency range with minimum and maximum values (`SamplingFrequencyRange`). All frequency values must be given in Hz.

17.3.4.3 Format III section

This format is used to transfer 16-bit stereo data (two channels).

```
FORMAT_III {
  BitResolution:          0int;
  //
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
  // The following items are valid for USB audio 1.0 devices only
  //
  SamplingFrequency:      0int ...;
  SamplingFrequencyRange: 0intMin 0intMax;
}
```

`BitResolution` specifies the number of effectively used bits from the available bits in an audio subframe / subslot. The value must be ≤ 16.

Supported sample frequencies can be specified either with a list of discrete frequencies (`SamplingFrequency`) or by a frequency range with minimum and maximum values (`SamplingFrequencyRange`). All frequency values must be given in Hz.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4 Target API

Function	Description
API functions	
USBD_AC_Add()	Adds an Audio interface to the USB stack.
USBD_AC_GetCurrentAltSetting()	Returns the current alternate setting of an interface, that was set by the host.
USBD_AC_GetStreamInfo()	Returns information about a streaming interface for a given alternate setting or for the current alternate setting, that was set by the host.
USBD_AC_OpenRXStream()	Opens an interface and starts reading audio data, depending of the current alternate setting, that was set by the host.
USBD_AC_CloseRXStream()	Stops data transfers of an audio receive stream and closes the stream.
USBD_AC_OpenTXStream()	Opens an interface and prepare for sending audio data with USBD_AC_Send(), depending of the current alternate setting, that was set by the host.
USBD_AC_Send()	Provide audio data to be sent to the host.
USBD_AC_CloseTXStream()	Stops data transfers of an audio send stream and closes the stream.
USBD_AC_SetFeedbackDataRate()	Provides sample rate feedback for an OUT endpoint using explicit asynchronous synchronization.
USBD_AC_GetFeedbackDataRate()	Gets the sample rate feedback that was sent by the host for an IN endpoint using explicit adaptive synchronization.
USBD_AC_SendInterruptMessage()	Writes an interrupt message via the optional interrupt IN endpoint to the host.
Data structures	
USBD_AC_INIT_DATA	Initialization data for the Audio class instance.
USBD_AC_STREAM_INTF_INFO	This structure contains information about an audio streaming interface.
USBD_AC_RX_CTX	Contains all information about an active interface receiving audio data.
USBD_AC_RX_DATA	This structure is used to forward audio data to the application.
USBD_AC_TX_CTX	Contains all information about an active interface for sending audio data.
USBD_AC_CONTROL_INFO	This structure contains information about the type of a control request.
USBD_AC_EVENT	Event types for RX / TX callback functions.
Function definitions	
USBD_AC_SET_ALT_INTERFACE	Definition of the callback which is called when the host sets an alternate setting on an audio interface.

 © 2010-2026 SEGGER Microcontroller GmbH

Function	Description
USBD_AC_CONTROL_GET_FUNC	Definition of the callback which is called when an audio control get request is received.
USBD_AC_CONTROL_SET_FUNC	Definition of the callback which is called when an audio control set request is received.
USBD_AC_RX_CALLBACK	Definition of the callback which is called when audio data was received from the host.
USBD_AC_TX_CALLBACK	Definition of the callback which is called when audio data was sent to the host.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1 API functions

17.4.1.1 USBD_AC_Add()

Description

Adds an Audio interface to the USB stack.

Prototype

```
int USBD_AC_Add(const USBD_AC_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a USBD_AC_INIT_DATA structure containing values for the initialization.

Return value

≥ 0 on success.
< 0 on error.

Additional information

This function registers Audio interface class with the USB stack. It must be called after USBD_Init() and before USBD_Start(). The structure USBD_AC_INIT_DATA has to be initialized before USBD_AC_Add() is called. Refer to USBD_AC_INIT_DATA for more information.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1.2 USBD_AC_GetCurrentAltSetting()

Description

Returns the current alternate setting of an interface, that was set by the host.

Prototype

```
unsigned USBD_AC_GetCurrentAltSetting(unsigned Interface);
```

Parameters

Parameter	Description
Interface	Index of the audio interface, see generated USBD_AC_INTERFACE_... defines.

Return value

Current alternate setting.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1.3 USBD_AC_GetStreamInfo()

Description

Returns information about a streaming interface for a given alternate setting or for the current alternate setting, that was set by the host.

Prototype

```
const USBD_AC_STREAM_INTF_INFO *USBD_AC_GetStreamInfo(unsigned Interface,
                                                      int      AltSetting);
```

Parameters

Parameter	Description
Interface	Index of the audio interface, see generated USBD_AC_INTERFACE_... defines.
AltSetting	Number of the alternate setting for which the information should be returned. Can be set to -1, then the current alternate setting selected by the host is used.

Return value

Pointer to a USBD_AC_STREAM_INTF_INFO structure containing the information.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1.4 USBD_AC_OpenRXStream()

Description

Opens an interface and starts reading audio data, depending of the current alternate setting, that was set by the host.

Prototype

```
int USBD_AC_OpenRXStream(USBD_AC_RX_CTX * pCtx);
```

Parameters

Parameter	Description
pCtx	Receive stream context. The public part of the structure must be initialized before calling USBD_AC_OpenRXStream(). After the stream was opened successfully, the data (memory area) must remain valid and must not be changed by the application.

Return value

= 0 Stream was successfully opened and receiving data has started.
≠ 0 Error.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1.5 USBD_AC_CloseRXStream()

Description

Stops data transfers of an audio receive stream and closes the stream.

Prototype

```
void USBD_AC_CloseRXStream(USBD_AC_RX_CTX * pCtx);
```

Parameters

Parameter	Description
pCtx	Receive context of a RX stream, that was successfully opened using USBD_AC_OpenRXStream(). After the function returns, the context is not used any more and can be destroyed.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1.6 USBD_AC_OpenTXStream()

Description

Opens an interface and prepare for sending audio data with USBD_AC_Send(), depending of the current alternate setting, that was set by the host.

Prototype

```
int USBD_AC_OpenTXStream(USBD_AC_TX_CTX * pCtx);
```

Parameters

Parameter	Description
pCtx	Send stream context. The public part of the structure must be initialized before calling USBD_AC_OpenTXStream(). After the stream was opened successfully, data can be sent with the function USBD_AC_Send().

Return value

= 0 Stream was successfully opened.
≠ 0 Error.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1.7 USBD_AC_Send()

Description

Provide audio data to be sent to the host. The caller needs to know the current sample rate and how the audio samples have to be distributed over the USB frames / micro frames.

This function returns immediately performing the data transfer asynchronously. After successful transfer of the data, the callback function provided in the USBD_AC_TX_CTX structure is called to indicate that new audio data can be accepted to be sent.

A maximum of two calls to USBD_AC_Send() can be queued. The send queue is used to support a double buffering of audio data: While a data packet is sent to the host, another data packet can be created by the application and queued with USBD_AC_Send().

Prototype

```
int USBD_AC_Send(      USBD_AC_TX_CTX * pCtx,
                       U16              NumFrames,
                       U32              NumBytes,
                 const void           * pData);
```

Parameters

Parameter	Description
pCtx	Send context of a TX stream, that was successfully opened using USBD_AC_OpenTXStream().
NumFrames	Number of frames (full-speed) or micro frames (high-speed) the provided data is used for. The data is equally distributed over this number of frames / micro frames. Must be ≤ 1024.
NumBytes	Number of bytes of the audio data. Must be ≤ NumFrames * MaxPacketSize. If NumFrames > 1, then NumBytes must be a multiple of USBD_AC_TX_CTX.SampleSize.
pData	Pointer to the audio data. To achieve best performance the data should be word aligned or cache aligned if the system is using a data cache.

Return value

= 0 Success.
≠ 0 Error.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1.8 USBD_AC_CloseTXStream()

Description

Stops data transfers of an audio send stream and closes the stream.

Prototype

```
void USBD_AC_CloseTXStream(USBD_AC_TX_CTX * pCtx);
```

Parameters

Parameter	Description
pCtx	Send context of a TX stream, that was successfully opened using USBD_AC_OpenTXStream(). After the function returns, the context is not used any more and can be destroyed.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1.9 USBD_AC_SetFeedbackDataRate()

Description

Provides sample rate feedback for an OUT endpoint using explicit asynchronous synchronization.

Prototype

```
void USBD_AC_SetFeedbackDataRate(USBD_AC_RX_CTX * pCtx,
                                 U32              DataRate);
```

Parameters

Parameter	Description
pCtx	Receive context of a RX stream, that was successfully opened using USBD_AC_OpenRXStream().
DataRate	Number of samples that are processed by the device within one interval (endpoint configuration) multiplied by 2^16. As the actual sample rate per interval may not be an integer, the multiplication with 2^16 results in a fixed point number, where the upper 16 bits are the integer part and the lower 16 bits contain the fractional part. Example: A data rate of 44.25 samples per interval is coded as DataRate = 0x002C4000.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1.10 USBD_AC_GetFeedbackDataRate()

Description

Gets the sample rate feedback that was sent by the host for an IN endpoint using explicit adaptive synchronization.

Prototype

```
U32 USBD_AC_GetFeedbackDataRate(USBD_AC_TX_CTX * pCtx);
```

Parameters

Parameter	Description
pCtx	Send context of a TX stream, that was successfully opened using USBD_AC_OpenTXStream().

Return value

DataRate: Number of samples that are processed by the device within one interval (endpoint configuration) multiplied by 2^16. As the actual sample rate per interval may not be an integer, the multiplication with 2^16 results in a fixed point number, where the upper 16 bits are the integer part and the lower 16 bits contain the fractional part. Example: A data rate of 44.25 samples per interval is coded as DataRate = 0x002C4000. The functions returns 0, if there was no feedback from the host so far.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.1.11 USBD_AC_SendInterruptMessage()

Description

Writes an interrupt message via the optional interrupt IN endpoint to the host.

Prototype

```
int USBD_AC_SendInterruptMessage(unsigned Interface,
                                 U32      ID,
                                 U8       ChannelNumber,
                                 int      ms);
```

Parameters

Parameter	Description
Interface	Index of the audio interface, use one of the generated USBD_AC_INTERFACE_... macros.
ID	Interrupt source: Unit / terminal / entity ID and audio control selector. Use one of the generated USBD_AC_ID_... macros added with one of the USB_AC_xx_..._CONTROL macros.
ChannelNumber	Channel number of the control unit or 0 if not used. Ignored for audio 1.0 devices.
ms	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function returns immediately and the transfer is processed asynchronously.

Return value

= 0 Successfully started an asynchronous write transfer (Timeout = -1) or a timeout has occurred and no data was written.
> 0 Write transfer successfully completed.
< 0 An error occurred.

Additional information

Endpoint related interrupt messages are not (yet) supported.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.2 Data structures

17.4.2.1 USBD_AC_INIT_DATA

Description

Initialization data for the Audio class instance.

Type definition

```
typedef struct {
  const USBD_AC_CONFIG        * pACConfig;
  USBD_AC_CONTROL_GET_FUNC    * pfControlGet;
  USBD_AC_CONTROL_SET_FUNC    * pfControlSet;
  USBD_AC_SET_ALT_INTERFACE   * pfSetAlternate;
  USBD_AC_GET_ALT_INTERFACE   * pfGetInterfaceInfo;
  U8                            IntEP;
} USBD_AC_INIT_DATA;
```

Structure members

Member	Description
pACConfig	Pointer to configuration data created by the generator.
pfControlGet	Callback function to handle audio control get requests.
pfControlSet	Callback function to handle audio control set requests.
pfSetAlternate	Callback to inform the application about Set Interface control requests.
pfGetInterfaceInfo	Callback to inform the application about Get Interface control requests.
IntEP	Optional interrupt EP. If used, it must be allocated by calling USBD_AddEP(1, USB_TRANSFER_TYPE_INT, Interval, NULL, PacketSize), where PacketSize must be 2 for audio 1.0 devices and 6 for audio 2.0 devices.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.2.2 USBD_AC_STREAM_INTF_INFO

Description

This structure contains information about an audio streaming interface.

Type definition

```
typedef struct {
  U16   MaxPacketSize;
  U8    IntervalExp;
  U8    NrChannels;
  U8    SubframeSize;
  U8    BitResolution;
} USBD_AC_STREAM_INTF_INFO;
```

Structure members

Member	Description
MaxPacketSize	Maximum packet size of the endpoint.
IntervalExp	Specifies the interval of the endpoint in micro frames (125us): Interval = 2^IntervalExp
NrChannels	Number of audio channels. If not specified in the audio description file, contains 0.
SubframeSize	Number of byte for a single audio sample (one channel). If not specified in the audio description file, contains 0.
BitResolution	Number of relevant bits in a single audio sample. If not specified in the audio description file, contains 0.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.2.3 USBD_AC_RX_CTX

Description

Contains all information about an active interface receiving audio data. The public part of this structure must be initialized by the application before passed to the function USBD_AC_OpenRXStream(). The data must remain valid and must not be modified by the application while the stream is open and used.

Type definition

```
typedef struct {
  U16                    Interface;
  U16                    Flags;
  void                (* pfSOFCallback)(void * pCtx);
  U16                    FeedbackInterval;
  USBD_AC_RX_CALLBACK  * pfCallback;
  USBD_AC_RX_DATA        RxData;
} USBD_AC_RX_CTX;
```

Structure members

Member	Description
Interface	Index of the audio interface, see generated USBD_AC_INTERFACE_... defines.
Flags	Reserved for future extensions, must be set to 0.
pfSOFCallback	Internal use.
FeedbackInterval	Interval (Number of SOFs) used for calling pfSOFCallback. Measured in units of 1ms for full-speed / 125us in high-speed.
pfCallback	Function that is called when data was received from the host or a 'close' event or timeout has occurred on the stream.
RxData	Data transfer information.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.2.4 USBD_AC_RX_DATA

Description

This structure is used to forward audio data to the application.

Type definition

```
typedef struct {
  void  * pBuffer;
  U32     NumBytes;
  U16     NumPackets;
  U16     Timeout;
  void  * pUserContext;
} USBD_AC_RX_DATA;
```

Structure members

Member	Description
pBuffer	Pointer to the buffer which is used to receive audio data. The buffer must be provided by the application and must have a size of at least the maximum packet size of the currently selected audio stream endpoint, rounded up to a multiple of 4 or a multiple of the cache line size of the system. To achieve best performance the buffer should be word aligned or cache aligned if the system is using a data cache. Must be initialized before calling USBD_AC_OpenRXStream() (within USBD_AC_RX_CTX) and in the pfCallback function before it returns.
NumBytes	Must be initialized by the application to the size of the buffer 'pBuffer'. When the function pfCallback is called, it contains the number of bytes actually received.
NumPackets	Must be initialized by the application to the maximum number of audio packets that are read into the buffer before the application is notified via the callback function. When the function pfCallback is called, it contains the number of packets actually received.
Timeout	Timeout in units of SOFs (1ms for full-speed / 125us in high-speed). If no packets were received within the specified time, the application is notified with a USBD_AC_EVENT_TIMOUT event. A value of 0 means no timeout.
pUserContext	Can be arbitrarily used by the application.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.2.5 USBD_AC_TX_CTX

Description

Contains all information about an active interface for sending audio data. The public part of this structure must be initialized by the application before passed to the function USBD_AC_OpenTXStream(). The data must remain valid and must not be modified by the application while the stream is open and used.

Type definition

```
typedef struct {
  U16                    Interface;
  U16                    Flags;
  U16                    SampleSize;
  U16                    Timeout;
  USBD_AC_TX_CALLBACK  * pfCallback;
  void                 * pUserContext;
} USBD_AC_TX_CTX;
```

Structure members

Member	Description
Interface	Index of the audio interface, see generated USBD_AC_INTERFACE_... defines.
Flags	Reserved for future extensions, must be set to 0.
SampleSize	Number of bytes for a single audio sample (all channels). The value is not required (and is ignored) if only a single audio packet is sent via USBD_AC_Send().
Timeout	Timeout in units of SOFs (1ms for full-speed / 125us in high-speed). If no packets were sent within the specified time, the application is notified with a USBD_AC_EVENT_TIMOUT event. A value of 0 means no timeout.
pfCallback	Function that is called when data was sent to the host
pUserContext	Can be arbitrarily used by the application.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.2.6 USBD_AC_CONTROL_INFO

Description

This structure contains information about the type of a control request.

Type definition

```
typedef struct {
  U32   ID;
  U8    bRequest;
  U8    ChannelNumber;
} USBD_AC_CONTROL_INFO;
```

Structure members

Member	Description
ID	Request ID: Bits 20..16 contain the interface index (0 = control interface, 1,… = streaming interfaces) Bits 15..8 contain the unit / terminal / entity ID. Is set to 0xFF if recipient is the endpoint. Bits 7..0 contain the control selector.
bRequest	Audio Class-Specific Request Code, see USB_AC_REQ… defines.
ChannelNumber	Internal use.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.2.7 USBD_AC_EVENT

Description

Event types for RX / TX callback functions.

Type definition

```
typedef enum {
  USBD_AC_EVENT_DATA_RECEIVED,
  USBD_AC_EVENT_DATA_SEND,
  USBD_AC_EVENT_TIMOUT,
  USBD_AC_EVENT_CLOSED
} USBD_AC_EVENT;
```

Enumeration constants

Constant	Description
USBD_AC_EVENT_DATA_RECEIVED	Data was read from the host.
USBD_AC_EVENT_DATA_SEND	Data was sent to the host.
USBD_AC_EVENT_TIMOUT	Timeout on read or write.
USBD_AC_EVENT_CLOSED	Interface was closed by the host.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.3 Function definitions

17.4.3.1 USBD_AC_SET_ALT_INTERFACE

Description

Definition of the callback which is called when the host sets an alternate setting on an audio interface. This callback is called in interrupt context and must not block.

Type definition

```
typedef void USBD_AC_SET_ALT_INTERFACE(unsigned InterfaceNo,
                                       unsigned NewAltSetting);
```

Parameters

Parameter	Description
InterfaceNo	Number of the audio streaming interface. Corresponds to the USBD_AC_INTERFACE_... defines.
NewAltSetting	Alternate setting selected by the host.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.3.2 USBD_AC_CONTROL_GET_FUNC

Description

Definition of the callback which is called when an audio control get request is received. This callback is called in interrupt context and must not block.

Type definition

```
typedef int USBD_AC_CONTROL_GET_FUNC(const USBD_AC_CONTROL_INFO * pReqInfo,
                                      U8                        * pBuffer);
```

Parameters

Parameter	Description
pReqInfo	Contains information about the type of the control request.
pBuffer	Pointer to a buffer into which the callback should write the reply (max. 64 bytes).

Return value

≥ 0 Audio control request was handled by the callback and response data was put into pBuffer. The callback function must return the length of the response data which will be sent to the host.

< 0 Audio control request was not handled by the callback (i.e. illegal request or parameters). The stack will STALL the request.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.3.3 USBD_AC_CONTROL_SET_FUNC

Description

Definition of the callback which is called when an audio control set request is received. This callback is called in interrupt context and must not block.

Type definition

```
typedef int USBD_AC_CONTROL_SET_FUNC(const USBD_AC_CONTROL_INFO * pReqInfo,
                                           U32                    NumBytes,
                                     const U8                   * pBuffer);
```

Parameters

Parameter	Description
pReqInfo	Contains information about the type of the control request.
NumBytes	Number of bytes in pBuffer.
pBuffer	Pointer to a buffer containing the request data.

Return value

= 0 Audio control request was handled by the callback.
≠ 0 Audio control request was not handled by the callback (i.e. illegal request or parameters). The stack will STALL the request.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.3.4 USBD_AC_RX_CALLBACK

Description

Definition of the callback which is called when audio data was received from the host. pRxData->NumBytes bytes of data were received into pRxData->pBuffer. The function must reinitialize the members pBuffer, NumBytes and MaxPackets before it returns. This callback is called in interrupt context and must not block. The audio data must not be processed inside this function, instead a task should be triggered that does the audio processing and this function should return as fast as possible. After this function has returned, the next USB transfer is started immediately. Therefore the member 'pBuffer' should be initialized to point to a different buffer to avoid overwriting the data just received (double buffering mechanism is recommended).

Type definition

```
typedef void USBD_AC_RX_CALLBACK(USBD_AC_EVENT     Event,
                                 USBD_AC_RX_DATA * pRxData);
```

Parameters

Parameter	Description
Event	Event occurred on the audio stream.
pRxData	Pointer to a USBD_AC_RX_DATA structure. The contents is valid only, if Event = USBD_AC_EVENT_DATA_RECEIVED.

 © 2010-2026 SEGGER Microcontroller GmbH

17.4.3.5 USBD_AC_TX_CALLBACK

Description

Definition of the callback which is called when audio data was sent to the host. The function should initiate to send more data.

Type definition

```
typedef void USBD_AC_TX_CALLBACK(      USBD_AC_EVENT Event,
                                 const void        * pData,
                                       void        * pUserContext);
```

Parameters

Parameter	Description
Event	Event occurred on the audio stream.
pData	Pointer to the data sent, which was provided to the USBD_AC_Send() function.
pContext	Pointer from the USBD_AC_RX_CTX structure.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 18

Legacy Audio 1.0

This chapter gives a general overview of the legacy Audio class and describes how to get the Audio component running on the target. If designing new audio applications, it’s recommended to use the new *Audio* class.



 © 2010-2026 SEGGER Microcontroller GmbH

18.1 Overview

> **Note**
>
> For new audio applications, the new *Audio* class should be used, even for Audio 1.0, see *Audio* on page 576.

The USB Audio device class is a USB class protocol which can be used to transfer sound data from a device to a host and vice versa.

Audio is supported by most operating systems out of the box, and the installation of additional drivers is not required.

emUSB-Device-Audio comes as a complete package and contains the following:

- Generic USB handling
- USB Audio V1 device class implementation
- Sample application showing how to work with Audio

 © 2010-2026 SEGGER Microcontroller GmbH

18.2 Introduction

SEGGER's implementation of the Audio class V1.0 is designed with minimal resource usage in mind, especially targeted at embedded devices. The implementation supports the usage of a "speaker" (input/output audio terminal with a feature terminal for controls) and a "microphone" (input/output audio terminal).

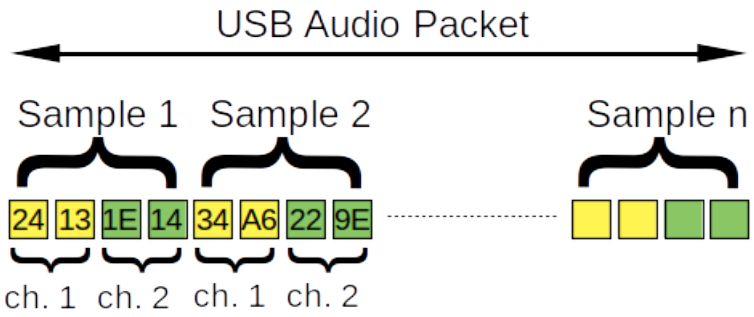
The speaker and microphone can be used independently of each other; both can be enabled at the same time, allowing audio transfer in either direction (headset-like operation).

The Audio class supports adaptive synchronization for OUT endpoints, asynchronous synchronization for IN endpoints, and synchronous synchronization for both directions. The commands `SET_CUR`, `GET_CUR`, `SET_MIN`, `GET_MIN`, `SET_MAX`, `GET_MAX`, `SET_RES`, and `GET_RES` are supported. For the speaker interface, feature unit controls are supported (volume, mute, etc.).

Warning

emUSB-Device-Audio does not provide drivers/codecs for any audio peripherals; writing a driver to interface with the audio hardware is the responsibility of the customer.

With emUSB-Device-Audio, Audio data is transferred in PCM encoding. The Audio class transfers multiple audio samples in a single packet. In the following sample the Audio class is configured with 2 channels (stereo) and 16-bit data per channel:



The length of a complete audio packet is equal to the bits per sample, rounded up to bytes, multiplied by the number of channels and the sample rate, then divided by 1000, as a packet is sent every millisecond. For a sample rate of 48000, 16 bits per sample, and 2 channels, the calculation is as follows:

48000 * 16/8 * 2 / 1000 = 192 bytes

For a sample rate of 44100, 16 bits per sample, and 2 channels, the calculation is as follows:

44100 * 16/8 * 2 / 1000 = 176.4 bytes

Since we cannot transfer 0.4 bytes, the audio packets need to be 176 bytes (44 samples) and each 10th packet (sample size divided by the remainder: 4 / 0.4) should contain 45 samples (180 bytes) to make sure the sample rate remains at 44100.

Note

On macOS (tested with Big Sur), when a device is using USB high-speed for an audio device, the speaker interface will not be shown, unless the ISO endpoint type of the OUT endpoint is "`USB_ISO_SYNC_TYPE_ASYNCHRONOUS`". Microphone interfaces are not affected.

 © 2010-2026 SEGGER Microcontroller GmbH

18.3 Configuration

18.3.1 Initial configuration

To get emUSB-Device-Audio up and running as well as doing an initial test, the configuration as delivered with the sample application should not be modified.

18.3.2 Final configuration

The configuration must only be modified when emUSB-Device is integrated in your final product. Refer to section *emUSB-Device Configuration* on page 53 for detailed information about the generic information functions which have to be adapted.

Windows

Windows systems save the audio settings for each USB Audio device inside the Windows registry (interfaces, number of channels, sub-frame-size, bit resolution and sample frequency). These values are saved for the USB Vendor ID and the USB Product ID. When a device with the same USB Vendor ID and USB Product ID enumerates a second time, the audio settings are checked against the saved values inside the Windows registry. If the settings do not match, the device will not function (Windows will not request any audio data from it).

It is not known why Windows behaves this way; other operating systems are not affected.

When developing a USB Audio device and experimenting with different sample rates, bit resolutions, etc., it is advised to remove the device from the registry after each change to the audio settings or to use a different USB Product ID after every change.

18.3.3 Using the microphone interface

When using the microphone sample applications with a PC it is not immediately clear whether they work as the PC only receives the audio data. To listen to the data being sent from the target running emUSB-Device-Audio to your PC it is necessary to enable a loopback mode which will transfer the audio data from the microphone interface to the physical speakers connected to your PC.

Linux

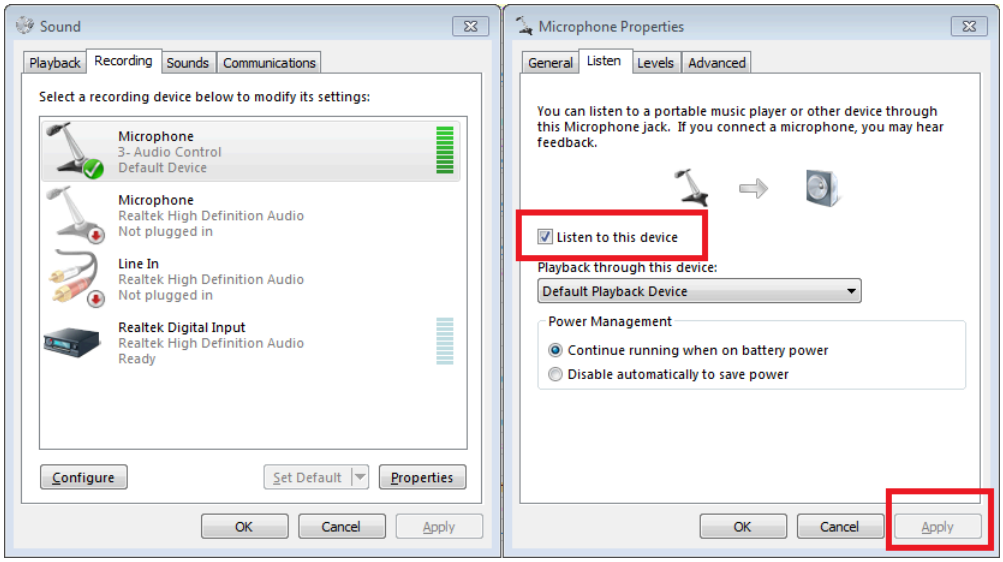
This guide assumes you are using PulseAudio.

- Make sure the device running the emUSB-Device-Audio microphone sample is selected as the default sound input device.
- Make sure your speakers (or headphones) are selected as the default sound output device.
- Run *pactl load-module module-loopback* to enable loopback.
- At this point you should hear the sound being produced by the microphone sample.
- You can run *pactl unload-module module-loopback* to disable the loopback mode.

Windows

- Make sure your speakers (or headphones) are selected as the default sound output device.
- In the sound configuration of the device running emUSB-Device-Audio microphone sample tick the “Listen to this device” checkbox and click “Apply”.
- At this point you should hear the sound being produced by the microphone sample.

 © 2010-2026 SEGGER Microcontroller GmbH



Mac

At the time of writing, no built-in way of looping back audio is known. But there are a couple of third-party applications out there which can enable loopback mode for macOS.

18.3.4 Using the speaker interface

When using the speaker sample applications, the PC merely needs to be configured to use the device running emUSB-Device-Audio as the default output sound device.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4 Target API

Function	Description
API functions	
USBD_AUDIO_Add()	Adds an Audio interface to the USB stack.
USBD_AUDIO_Read_Task()	Task function of the Audio component which processes data received from host.
USBD_AUDIO_Write_Task()	Task function of the Audio component which processes data sent to the host.
USBD_AUDIO_Start_Play()	Starts providing audio data to the host using the microphone terminal of the audio class.
USBD_AUDIO_Stop_Play()	Stops providing audio data to the host.
USBD_AUDIO_Start_Listen()	Starts receiving audio data from the host using the speaker terminal of the audio class.
USBD_AUDIO_Stop_Listen()	Stops receiving audio data from the host.
USBD_AUDIO_Set_Timeouts()	Sets the timeouts used by the USBD_AUDIO_Read_Task() and USBD_AUDIO_Write_Task() when listen/play is active.
Data structures	
USBD_AUDIO_INIT_DATA	Initialization data for the Audio class instance.
USBD_AUDIO_IF_CONF	Initialization structure for an audio microphone/speaker interface.
USBD_AUDIO_FORMAT	Initialization data for a single audio format.
USBD_AUDIO_UNITS	This structure contains IDs used for a particular audio interface.
Function definitions	
USBD_AUDIO_TX_FUNC	Definition of the callback which is called when audio data is sent.
USBD_AUDIO_RX_FUNC	Definition of the callback which is called when audio data is received.
USBD_AUDIO_CONTROL_FUNC	Definition of the callback which is called when audio commands are received.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.1 API functions

18.4.1.1 USBD_AUDIO_Add()

Description

Adds an Audio interface to the USB stack.

Prototype

```
USBD_AUDIO_HANDLE USBD_AUDIO_Add(const USBD_AUDIO_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a USBD_AUDIO_INIT_DATA structure containing values for the initialization. This structure's memory must remain available while the audio class is being used. The application should not put this structure on the stack.

Return value

USBD_AUDIO_HANDLE - Handle for the added Audio instance.

Additional information

After the initialization of USB core, this is the first function that needs to be called when an Audio interface is used with emUSB-Device. The structure USBD_AUDIO_INIT_DATA has to be initialized before USBD_AUDIO_Add() is called. Refer to USBD_AUDIO_INIT_DATA for more information.

For the Audio component to be functional one or both of the following functions have to be created as a task: USBD_AUDIO_Read_Task(), USBD_AUDIO_Write_Task().

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.1.2 USBD_AUDIO_Read_Task()

Description

Task function of the Audio component which processes data received from host. Handles operations of the speaker interface. Has to be created as a separate task.

Prototype

```
void USBD_AUDIO_Read_Task(void);
```

Additional information

Only necessary if the speaker interface is used. The function returns only when `USBD_DeInit()` is called.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.1.3 USBD_AUDIO_Write_Task()

Description

Task function of the Audio component which processes data sent to the host. Handles operations of the microphone interface. Has to be created as a separate task.

Prototype

```
void USBD_AUDIO_Write_Task(void);
```

Additional information

Only necessary if the microphone interface is used. The function returns only when `USBD_DeInit()` is called.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.1.4 USBD_AUDIO_Start_Play()

Description

Starts providing audio data to the host using the microphone terminal of the audio class.

Prototype

```
int USBD_AUDIO_Start_Play(      USBD_AUDIO_HANDLE   hInst,
                          const U8                * pBufIn);
```

Parameters

Parameter	Description
hInst	Handle to a valid Audio instance, returned by USBD_AUDIO_Add().
pBufIn	Buffer initially used with IN transfers (microphone interface). Can be changed inside the USBD_AUDIO_TX_FUNC callback.

Return value

= 0 Success.
< 0 An error occurred.

Additional information

This function enables the registered TX user function (USBD_AUDIO_TX_FUNC). The callback is called after every successful transfer and should move the buffer pointer to the next audio packet accordingly or fill the same buffer with new data. The callback is called in an interrupt context. The execution of the callback together with the internal routines must never take longer than 1 millisecond because the audio class must send one packet every millisecond.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.1.5 USBD_AUDIO_Stop_Play()

Description

Stops providing audio data to the host.

Prototype

```
void USBD_AUDIO_Stop_Play(USBD_AUDIO_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid Audio instance, returned by USBD_AUDIO_Add().

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.1.6 USBD_AUDIO_Start_Listen()

Description

Starts receiving audio data from the host using the speaker terminal of the audio class.

Prototype

```
int USBD_AUDIO_Start_Listen(USBD_AUDIO_HANDLE   hInst,
                            U8                * pBufOut);
```

Parameters

Parameter	Description
hInst	Handle to a valid Audio instance, returned by USBD_AUDIO_Add().
pBufOut	Buffer initially used with OUT transfers (speaker interface). Can be changed inside the USBD_AUDIO_RX_FUNC callback.

Return value

= 0 Success.
< 0 An error occurred.

Additional information

This function enables the registered user callback function (USBD_AUDIO_RX_FUNC) which is called before the host sends data to the target. Inside the callback you may read the received data. The callback is called in an interrupt context. The execution of the callback together with the internal routines must never take longer than 1 millisecond because the audio class must send one packet every millisecond.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.1.7 USBD_AUDIO_Stop_Listen()

Description

Stops receiving audio data from the host.

Prototype

```
void USBD_AUDIO_Stop_Listen(USBD_AUDIO_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid Audio instance, returned by USBD_AUDIO_Add().

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.1.8 USBD_AUDIO_Set_Timeouts()

Description

Sets the timeouts used by the USBD_AUDIO_Read_Task() and USBD_AUDIO_Write_Task() when listen/play is active.

Prototype

```
void USBD_AUDIO_Set_Timeouts(USBD_AUDIO_HANDLE hInst,
                             unsigned          ReadTimeout,
                             unsigned          WriteTimeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid Audio instance, returned by USBD_AUDIO_Add().
ReadTimeout	Read (OUT) timeout in milliseconds.
WriteTimeout	Write (IN) timeout in milliseconds.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.2 Data structures

18.4.2.1 USBD_AUDIO_INIT_DATA

Description

Initialization data for the Audio class instance.

Type definition

```
typedef struct {
  U8                          EPIn;
  U8                          EPOut;
  unsigned                    OutPacketSize;
  USBD_AUDIO_RX_FUNC        * pfOnOut;
  USBD_AUDIO_TX_FUNC        * pfOnIn;
  USBD_AUDIO_CONTROL_FUNC   * pfOnControl;
  void                      * pControlUserContext;
  U8                          NumInterfaces;
  const USBD_AUDIO_IF_CONF  * paInterfaces;
  void                      * pOutUserContext;
  void                      * pInUserContext;
} USBD_AUDIO_INIT_DATA;
```

Structure members

Member	Description
EPIn	Isochronous IN endpoint for sending data to the host. If microphone functionality is not desired set this to 0.
EPOut	Isochronous OUT endpoint for receiving data from the host. If speaker functionality is not desired set this to 0.
OutPacketSize	Size of a single audio OUT packet. Must be calculated as follows: (highest used) SampleRate * NumChannels * BitsPerSample / 8 / 1000
pfOnOut	Pointer to a function of type USBD_AUDIO_RX_FUNC which handles incoming audio data. Needs to be set when the speaker interface is used.
pfOnIn	Pointer to a function of type USBD_AUDIO_TX_FUNC which handles outgoing audio data. Needs to be set when the microphone interface is used.
pfOnControl	Pointer to a function of type USBD_AUDIO_CONTROL_FUNC which handles audio commands. Always needs to be set.
pControlUserContext	Pointer to a user context which is passed to the pfOnControl function. Optional, can be NULL.
NumInterfaces	Number of elements in the paInterfaces array.
paInterfaces	Pointer to an array of structures of type USBD_AUDIO_IF_CONF which contain configuration data for the audio interfaces.
pOutUserContext	Pointer to a user context which is passed to the pfOnOut function. Optional, can be NULL.
pInUserContext	Pointer to a user context which is passed to the pfOnIn function. Optional, can be NULL.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.2.2 USBD_AUDIO_IF_CONF

Description

Initialization structure for an audio microphone/speaker interface. Only one speaker and one microphone is supported.

Type definition

```
typedef struct _USBD_AUDIO_IF_CONF {
  U8                          Flags;
  U8                          Controls;
  U8                          TotalNrChannels;
  U8                          NumFormats;
  const USBD_AUDIO_FORMAT   * paFormats;
  U16                         bmChannelConfig;
  U16                         TerminalType;
  USBD_AUDIO_UNITS          * pUnits;
} USBD_AUDIO_IF_CONF;
```

Structure members

Member	Description
Flags	Reserved. Set to zero.
Controls	Bitmask, a bit set to 1 indicates that the mentioned Control is supported: • b0: Mute • b1: Volume • b2: Bass • b3: Mid • b4: Treble • b5: Graphic Equalizer • b6: Automatic Gain • b7: Delay
TotalNrChannels	Number of audio channels for this interface.
NumFormats	Number of elements inside the paFormats array.
paFormats	Pointer to any array of USBD_AUDIO_FORMAT structures.
bmChannelConfig	Bit map indicating the spatial locations of channels. Important: this value should not be left at 0 to avoid an issue with Windows. The bits correspond to the following locations: • b0: Left Front (L) • b1: Right Front (R) • b2: Center Front (C) • b3: Low Frequency Enhancement (LFE) • b4: Left Surround (LS) • b5: Right Surround (RS) • b6: Left of Center (LC) • b7: Right of Center (RC) • b8: Surround (S) • b9: Side Left (SL) • b10: Side Right (SR) • b11: Top (T) • b15..12: Reserved Channels are assigned to locations in ascending order. E.g. if b6 and b11 are set and the other bits are zero channel 0 will be “LC” and channel 1 will be “T”. Having more channels than bits set in this bit map is valid, the channels which do not have a bit set will be considered to have a non-predefined spatial position.

 © 2010-2026 SEGGER Microcontroller GmbH

Member	Description
TerminalType	Defines the type of speaker/microphone for this interface. Only one speaker and one microphone is supported! The following defines can be used: • USB_AUDIO_TERMTYPE_INPUT_UNDEFINED • USB_AUDIO_TERMTYPE_INPUT_MICROPHONE • USB_AUDIO_TERMTYPE_INPUT_DESKTOP_MICROPHONE • USB_AUDIO_TERMTYPE_INPUT_PERSONAL_MICROPHONE • USB_AUDIO_TERMTYPE_INPUT_OMNI_DIRECTIONAL_MICROPHONE • USB_AUDIO_TERMTYPE_INPUT_MICROPHONE_ARRAY • USB_AUDIO_TERMTYPE_INPUT_PROCESSING_MICROPHONE_ARRAY • USB_AUDIO_TERMTYPE_OUTPUT_UNDEFINED • USB_AUDIO_TERMTYPE_OUTPUT_SPEAKER • USB_AUDIO_TERMTYPE_OUTPUT_HEADPHONES • USB_AUDIO_TERMTYPE_OUTPUT_HEAD_MOUNTED_DISPLAY_AUDIO • USB_AUDIO_TERMTYPE_OUTPUT_DESKTOP_SPEAKER • USB_AUDIO_TERMTYPE_OUTPUT_ROOM_SPEAKER • USB_AUDIO_TERMTYPE_OUTPUT_COMMUNICATION_SPEAKER • USB_AUDIO_TERMTYPE_OUTPUT_LOW_FREQUENCY_EFFECTS_SPEAKER
pUnits	Pointer to a structure of type USBD_AUDIO_UNITS. This structure is filled by the emUSB-Device Audio class during initialization.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.2.3 USBD_AUDIO_FORMAT

Description

Initialization data for a single audio format.

Type definition

```
typedef struct _USBD_AUDIO_FORMAT {
  U8    Flags;
  U8    NrChannels;
  U8    SubFrameSize;
  U8    BitResolution;
  U32   SamFreq;
} USBD_AUDIO_FORMAT;
```

Structure members

Member	Description
Flags	Reserved. Set to zero.
NrChannels	Number of channels in this format. Must never be greater than USBD_AUDIO_IF_CONF->TotalNrChannels While it is possible to configure less than the total number of channels
SubFrameSize	Size of an audio frame in bytes. Must be able to hold BitResolution bits.
BitResolution	Number of bits inside the audio frame dedicated to audio data. (Any remaining bits are padding.)
SamFreq	Supported sample frequency in Hz.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.2.4 USBD_AUDIO_UNITS

Description

This structure contains IDs used for a particular audio interface. The application should leave those values at zero, they are set by the stack after USBD_AUDIO_Add() has been called.

Type definition

```
typedef struct _USBD_AUDIO_UNITS {
  U8   Flags;
  U8   InterfaceNo;
  U8   AltInterfaceNo;
  U8   InputTerminalID;
  U8   OutputTerminalID;
  U8   FeatureUnitID;
} USBD_AUDIO_UNITS;
```

Structure members

Member	Description
Flags	Reserved. Set to zero.
InterfaceNo	USB Interface number of the audio interface. Set by the emUSB-Device stack.
AltInterfaceNo	Alternate setting number of the USB interface. Set by the emUSB-Device stack.
InputTerminalID	ID of the input terminal. Set by the emUSB-Device stack.
OutputTerminalID	ID of the output terminal. Set by the emUSB-Device stack.
FeatureUnitID	ID of the feature unit. Set by the emUSB-Device stack.

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.3 Function definitions

18.4.3.1 USBD_AUDIO_TX_FUNC

Description

Definition of the callback which is called when audio data is sent. This callback is called in the context of `USBD_AUDIO_Write_Task()`

Type definition

```
typedef void USBD_AUDIO_TX_FUNC(      void * pUserContext,
                                const U8 * * ppNextBuffer,
                                      U32  * pNextPacketSize);
```

Parameters

Parameter	Description
pUserContext	User context which is passed to the callback.
ppNextBuffer	Buffer containing audio samples which should match the configuration from microphone `USBD_AUDIO_IF_CONF`. Initially this points to the `pBufIn` from the call to `USBD_AUDIO_Start_Play` function. The user can change this pointer to a different buffer which will be used in the next transaction or fill the same buffer with new data.
pNextBufferSize	Size of the next buffer.

Example

```
static void _cbOnIn(void * pUserContext,
                    const U8 ** ppNextBuffer,
                    U32 * pNextPacketSize) {
  USB_MEMCPY(_pBufMic, _pDataSource, PACKET_SIZE_IN);
  *ppNextBuffer    = _pBufMic;
  *pNextPacketSize = PACKET_SIZE_IN;
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.3.2 USBD_AUDIO_RX_FUNC

Description

Definition of the callback which is called when audio data is received. This callback is called in the context of `USBD_AUDIO_Read_Task()`. The default timeout is 50 milliseconds.

Type definition

```
typedef void USBD_AUDIO_RX_FUNC(void * pUserContext,
                                int    NumBytesReceived,
                                U8 * * ppNextBuffer,
                                U32  * pNextBufferSize);
```

Parameters

Parameter	Description
`pUserContext`	User context which is passed to the callback.
`NumBytesReceived`	The number of bytes which have been read in this transaction.
`ppNextBuffer`	Buffer containing audio samples which should match the configuration from speaker `USBD_AUDIO_IF_CONF`. Initially this points to the `pBufOut` from the `USBD_AUDIO_INIT_DATA` structure. The user can change this pointer to a different buffer which will be used in the next transaction or leave it as it is and copy the data from this buffer elsewhere.
`pNextBufferSize`	Size of the next buffer.

 © 2010-2026 SEGGER Microcontroller GmbH

Example

```
static U8   _acBuf1[BUFFER_SIZE];
static U8   _acBuf2[BUFFER_SIZE];
static U8 * _pBuf;
static U8 * _pBufComplete;
static U32  _NumBytesInFullBuffer;
static U32  _NumBytesInBuffer;

// Receive callback function.
static void _cbOnOut(void * pUserContext,
                     int NumBytesReceived,
                     U8 ** ppNextBuffer,
                     U32 * pNextBufferSize) {
  char MBEvent;

  //
  // Check if the _next_ transfer would still fit into the buffer.
  // If not - switch the buffer.
  //
  if ((_NumBytesInBuffer + NumBytesReceived + PACKET_SIZE_OUT) >  BUFFER_SIZE) {
    //
    // Switch buffers.
    //
    if (_CurrentBuffer == 1) {
      _CurrentBuffer = 2;
      _pBuf = _acBuf2;
    } else {
      _CurrentBuffer = 1;
      _pBuf = _acBuf1;
    }
    _NumBytesInFullBuffer = _NumBytesInBuffer + NumBytesReceived;
    _NumBytesInBuffer = 0;
    MBEvent = BUFFER_FULL;
    //
    // Notify the task that a buffer is full.
    //
    if (OS_PutMailCond1(&_MailBox, &MBEvent) != 0) {
      printf("Missed packet.");
    }
  } else {
    _pBuf += NumBytesReceived;
    _NumBytesInBuffer += NumBytesReceived;
  }
  *ppNextBuffer = _pBuf;
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

18.4.3.3 USBD_AUDIO_CONTROL_FUNC

Description

Definition of the callback which is called when audio commands are received. This callback is called in an interrupt context.

Type definition

```
typedef int USBD_AUDIO_CONTROL_FUNC(void * pUserContext,
                                    U8     Event,
                                    U8     Unit,
                                    U8     ControlSelector,
                                    U8   * pBuffer,
                                    U32    NumBytes,
                                    U8     InterfaceNo,
                                    U8     AltSetting);
```

Parameters

Parameter	Description
pUserContext	User context which is passed to the callback.
Event	Audio event ID.
Unit	ID of the feature unit. In case of USB_AUDIO_PLAYBACK_* and USB_AUDIO_RECORD_*: 0.
ControlSelector	ID of the control. In case of USB_AUDIO_PLAYBACK_* and USB_AUDIO_RECORD_*: 0.
pBuffer	In case of GET events: pointer to a buffer into which the callback should write the reply. In case of SET events: pointer to a buffer containing the command value. In case of USB_AUDIO_PLAYBACK_* and USB_AUDIO_RECORD_*: NULL.
NumBytes	In case of GET events: requested size of the reply in bytes. In case of SET events: number of bytes in pBuffer. In case of USB_AUDIO_PLAYBACK_* and USB_AUDIO_RECORD_*: 0.
InterfaceNo	The number of the USB interface for which the event was issued.
AltSetting	The alternative setting number of the USB interface for which the event was issued.

Return value

= 0 Audio command was handled by the callback. The stack will send the reply.
≠ 0 Audio command was not handled by the callback. The stack will STALL the request.

Additional information

USB_AUDIO_PLAYBACK_* & USB_AUDIO_RECORD_* events are sent upon receiving a Set Interface USB request for Alternate Setting 1 for the respective interface (microphone or speaker). By default an Audio interface is set to Alternative Setting 0 in which it can not send or receive anything. The host switches the Alternative Setting to 1 when it has to send data to the device, this can be e.g. triggered by pressing “play” in your music player. Normally the host should switch the device back to Alternative Interface 0 when it has stopped sending audio data. This works well on Linux and OS X, but does not work reliably on Windows. When using Windows as a host it seems to depend on the application whether these events are generated or not. E.g. with some applications you will receive USB_AUDIO_PLAYBACK_START when “play” is pressed, but USB_AUDIO_PLAYBACK_STOP will not be sent when “pause” or “stop” is pressed. Relying on these events to check when the host has stopped sending data is not advised, instead set timeouts via USBD_AUDIO_Set_Timeouts and check for timeouts inside your USBD_AUDIO_RX_FUNC and USBD_AUDIO_TX_FUNC .

 © 2010-2026 SEGGER Microcontroller GmbH

 © 2010-2026 SEGGER Microcontroller GmbH

Example

```
// Control callback function.
static int _cbOnControl(void * pUserContext,
                        U8 Event,
                        U8 Unit,
                        U8 ControlSelector,
                        U8 * pBuffer,
                        U32 NumBytes) {
  int r;

  r = 0;
  switch (Event) {
  case USB_AUDIO_SET_CUR:
    switch (ControlSelector) {
      case USB_AUDIO_MUTE_CONTROL:
        if (*pBuffer == 1) {
          _SetMute(1);
        } else {
          _SetMute(0);
        }
        break;
      default:
        r = 1;
        break;
    }
    break;
  <...>
  <handle other commands>
  <...>
  }
  return r;
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 19

USB video device class (UVC)

This chapter gives a general overview of the UVC class and describes how to get the UVC component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

19.1 Overview

The USB video device class (UVC) is a USB class protocol which can be used to transfer video data from a device to a host.

UVC is supported by most operating systems out of the box and the installation of additional drivers is not required.

emUSB-Device-UVC comes as a complete package and contains the following:

- Generic USB handling
- USB video device class implementation
- Sample application showing how to work with UVC

 © 2010-2026 SEGGER Microcontroller GmbH

19.2 Configuration

19.2.1 Initial configuration

To get emUSB-Device-UVC up and running as well as to do an initial test, the configuration as delivered with the sample application should not be modified.

19.2.1.1 Uncompressed video format

Video data is transmitted using emUSB-Device Video in the uncompressed format. YUYV422 is the specific format used by the USB video device class. The format uses luminance (the brightness) and chrominance (the coloration) to display pictures. This is best explained by taking a look at the actual data; the first 8 bytes of a YUYV422 frame are defined as follows:

- 1 byte luminance of the first pixel (Y)
- 1 byte chrominance (blue) of the first and second pixel (U)
- 1 byte luminance of the second pixel (Y)
- 1 byte chrominance (red) of the first and second pixel (V)
- 1 byte luminance of the third pixel (Y)
- 1 byte chrominance (blue) of the third and fourth pixel (U)
- 1 byte luminance of the fourth pixel (Y)
- 1 byte chrominance (red) of the third and fourth pixel (V)

Using 1 byte for the chrominance of two pixels allows this format to save a byte per pixel when compared to the common RGB format (2 pixels YUYV422 - 4 bytes, 2 pixels RGB888 - 6 bytes).

Data must be provided in the YUYV422 format when using `USBD_UVC_Write()` or `USBD_UVC_WriteEx()`.

19.2.2 Final configuration

The configuration must only be modified when emUSB-Device is integrated into your final product. Refer to section *emUSB-Device Configuration* on page 53 for detailed information about the generic information functions which have to be adapted.

 © 2010-2026 SEGGER Microcontroller GmbH

19.3 Target API

Function	Description
API functions	
USBD_UVC_Add()	Adds a UVC interface to the USB stack.
USBD_UVC_Write()	Writes frame data to the host.
USBD_UVC_WriteEx()	Writes frame data to the host using single packets.
USBD_UVC_SetOnResolutionChange()	Allows to set a callback which is called when the host changes the resolution of the UVC frame.
Data structures	
USBD_UVC_INIT_DATA	Initialization data for UVC interface.
USBD_UVC_BUFFER	Structure which contains information about the UVC ring buffer.
USBD_UVC_DATA_BUFFER	Structure which contains values for a single buffer.
USBD_UVC_RESOLUTION	Structure describing a valid image resolution.
Function prototypes	
USB_UVC_ON_RESOLUTION_CHANGE	Callback function description which is set via USBD_UVC_SetOnResolutionChange().

 © 2010-2026 SEGGER Microcontroller GmbH

19.3.1 API functions

19.3.1.1 USBD_UVC_Add()

Description

Adds a UVC interface to the USB stack.

Prototype

```
int USBD_UVC_Add(const USBD_UVC_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to a USBD_UVC_INIT_DATA structure containing values for the initialization of the UVC module.

Return value

0 - Successfully added. 1 - An error occurred.

Additional information

After the initialization of USB core, this is the first function that needs to be called when a UVC interface is used with emUSB-Device. The structure USBD_UVC_INIT_DATA has to be initialized before USBD_UVC_Add() is called. Refer to USBD_UVC_INIT_DATA for more information.

 © 2010-2026 SEGGER Microcontroller GmbH

19.3.1.2 USBD_UVC_Write()

Description

Writes frame data to the host.

Prototype

```
int USBD_UVC_Write(const U8       * pData,
                   unsigned         NumBytes,
                   U8               UserFlags);
```

Parameters

Parameter	Description
pData	Pointer to a buffer containing the frame data.
NumBytes	Number of bytes in the buffer.
UserFlags	Flags to be added to the frame. Following flags are currently supported: • USBD_UVC_END_OF_FRAME - Should be set with the last USBD_UVC_Write() call for a single frame.

Return value

0 - All data written to the buffer.

−1 An error occurred (device disconnected).

Additional information

It is up to the application how much data it provides through this function, but providing a buffer containing a whole video frame will cause the least overhead. The application has to set the flag USBD_UVC_END_OF_FRAME when the last data part of a frame was written via USBD_UVC_Write(). Internally this function will write data into the buffers which have been initialized by the call to USBD_UVC_Add(). This allows for the buffers to be filled with video data before data is requested by the host application. The data transmission itself happens inside an interrupt triggered event callback inside the UVC module.

With every transmission the UVC module must add a payload header to the transfer. Therefore if the application needs to achieve maximum throughput the application should write MaxPacketSize - USBD_UVC_PAYLOAD_HEADER_SIZE chunks.

Do not mix usage of USBD_UVC_Write() and USBD_UVC_WriteEx().

Example

Sample describing a write operation where a frame is entirely available in a single buffer:

```
USBD_UVC_Write(h, WholeFrame, sizeof(WholeFrame), USBD_UVC_END_OF_FRAME);
```

Sample describing a write operation where a frame is only available in chunks:

```
U32 NumBytesAtOnce;
U32 NumBytesTotal;
U8 Flags;

NumBytesTotal = 153600; // Fixed frame size.
NumBytesAtOnce = SEGGER_MIN(sizeof(SmallBuffer), NumBytesTotal);
Flags = 0;
while (NumBytesTotal) {
 USBD_UVC_Write(h, SmallBuffer, NumBytesAtOnce, Flags);
 NumBytesTotal -= NumBytesAtOnce;
 NumBytesAtOnce = SEGGER_MIN(sizeof(SmallBuffer), NumBytesTotal);
 if (NumBytesTotal <= sizeof(SmallBuffer)) {
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
    Flags = USBD_UVC_END_OF_FRAME; // This will be the last write for this frame.
  }
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

19.3.1.3 USBD_UVC_WriteEx()

Description

Writes frame data to the host using single packets.

Prototype

```
int USBD_UVC_WriteEx(U8       * pData,
                     unsigned   NumBytes,
                     U8         UserFlags);
```

Parameters

Parameter	Description
pData	Pointer to a buffer containing the frame data. The buffer must provide USBD_UVC_PAYLOAD_HEADER_SIZE bytes space at the start of the buffer.
NumBytes	Size of the buffer. Must only contain one packet, up to a maximum size of USB_HS_ISO_HB_MAX_PACKET_SIZE.
UserFlags	Flags to be added to the frame. Following flags are currently supported: • USBD_UVC_END_OF_FRAME - Should be set with the last USBD_UVC_Write() call for a single frame.

Return value

0	All data written to the buffer.
−1	An error occurred (device disconnected).

Additional information

This version of the write routine is optimized to be used with DMA capable targets. But can also speed up transfers with regular drivers. This write routine does not copy the UVC packet data internally, but sends it from the user buffer directly. When using DMA and cache the buffer should be aligned to a cache line boundary. The buffer must provide space (USBD_UVC_PAYLOAD_HEADER_SIZE bytes) at the start of the buffer for the UVC module to insert the UVC packet header. The application must insert the video data after the header. The UVC module will automatically insert the correct header information and send the whole buffer to the host.

 © 2010-2026 SEGGER Microcontroller GmbH

19.3.1.4 USBD_UVC_SetOnResolutionChange()

Description

Allows to set a callback which is called when the host changes the resolution of the UVC frame. The callback receives a frame index, which is a direct mapping of the aResolutions array in USBD_UVC_INIT_DATA.

Prototype

```
void USBD_UVC_SetOnResolutionChange(USB_UVC_ON_RESOLUTION_CHANGE * pfOnResChange);
```

Parameters

Parameter	Description
pfOnResChange	User callback of type USB_UVC_ON_RESOLUTION_CHANGE .

 © 2010-2026 SEGGER Microcontroller GmbH

19.3.2 Data structures

19.3.2.1 USBD_UVC_INIT_DATA

Description

Initialization data for UVC interface.

Type definition

```
typedef struct _USBD_UVC_INIT_DATA {
  U8                            EPIn;
  USBD_UVC_BUFFER             * pBuf;
  const USBD_UVC_RESOLUTION   * aResolutions;
  U8                            NumResolutions;
  U8                            StillCaptureMethod;
  U8                            Flags;
  USBD_UVC_CONTROLS           * Controls;
} USBD_UVC_INIT_DATA;
```

Structure members

Member	Description
EPIn	Isochronous IN endpoint for sending data to the host.
pBuf	Pointer to a USBD_UVC_BUFFER structure.
aResolutions	Pointer to an array of USBD_UVC_RESOLUTION structures.
NumResolutions	Number of elements inside the aResolutions array.
StillCaptureMethod	Method of “still image capture” to use. Valid values: • 1 - The host software will extract the next available video frame. (default) • 2 - When the host requests a still image a callback will be called which has to provide a new (still) image frame. It only makes sense to use this method if your data source is able to provide better quality still images than the default quality of the video stream.
Flags	Various flags. Valid bits: • USBD_UVC_USE_BULK_MODE - In this mode UVC uses bulk endpoints instead of isochronous endpoints.
Controls	Pointer to a structure of type USBD_UVC_CONTROLS. The structure memory must remain available to the UVC class.

Additional information

This structure holds the endpoint that should be used by the UVC interface (EPIn). Refer to USBD_AddEPEx() for more information about how to add an endpoint.

 © 2010-2026 SEGGER Microcontroller GmbH

19.3.2.2 USBD_UVC_BUFFER

Description

Structure which contains information about the UVC ring buffer.

Type definition

```
typedef struct _USBD_UVC_BUFFER {
  USBD_UVC_DATA_BUFFER   Buf[USBD_UVC_NUM_BUFFERS];
  volatile U8            NumBlocksIn;
  U8                     RdPos;
  U8                     WrPos;
  U8                     Flags;
} USBD_UVC_BUFFER;
```

Structure members

Member	Description
Buf	Array of USBD_UVC_DATA_BUFFER elements.
NumBlocksIn	Number of currently used buffers.
RdPos	Buffer read position.
WrPos	Buffer write position.
Flags	Used by the UVC module automatically. Do not modify. 1 - WriteEx used.

Additional information

The number of buffers can be set with the USBD_UVC_NUM_BUFFERS define. Generally the user does not have to interact with this structure, but he has to provide the memory for it. When USBD_UVC_USE_BULK_MODE is used USBD_UVC_NUM_BUFFERS can be reduced to 1.

 © 2010-2026 SEGGER Microcontroller GmbH

19.3.2.3 USBD_UVC_DATA_BUFFER

Description

Structure which contains values for a single buffer.

Type definition

```
typedef struct _USBD_UVC_DATA_BUFFER {
  U8       * pData;
  unsigned   NumBytesIn;
  U8         Flags;
  U8         FrameID;
} USBD_UVC_DATA_BUFFER;
```

Structure members

Member	Description
pData	Pointer to a data buffer. When USBD_UVC_Write() is used the user must set this pointer to a valid buffer of size USBD_UVC_DATA_BUFFER_SIZE. When USBD_UVC_WriteEx() is used the user must not modify this value.
NumBytesIn	Size of the packet.
Flags	Flags which will be sent with the packet.
FrameID	ID of the frame.

Additional information

The size of the buffers can be set with the USBD_UVC_DATA_BUFFER_SIZE define. Ideally it should match the MaxPacketSize for the isochronous endpoint.

 © 2010-2026 SEGGER Microcontroller GmbH

19.3.2.4 USBD_UVC_RESOLUTION

Description

Structure describing a valid image resolution.

Type definition

```
typedef struct _USBD_UVC_RESOLUTION {
  unsigned   Width;
  unsigned   Height;
} USBD_UVC_RESOLUTION;
```

Structure members

Member	Description
Width	Width in pixels.
Height	Height in pixels.

 © 2010-2026 SEGGER Microcontroller GmbH

19.3.3 Function prototypes

19.3.3.1 USB_UVC_ON_RESOLUTION_CHANGE

Description

Callback function description which is set via `USBD_UVC_SetOnResolutionChange()`.

Type definition

```
typedef void USB_UVC_ON_RESOLUTION_CHANGE(unsigned FrameIndex);
```

Parameters

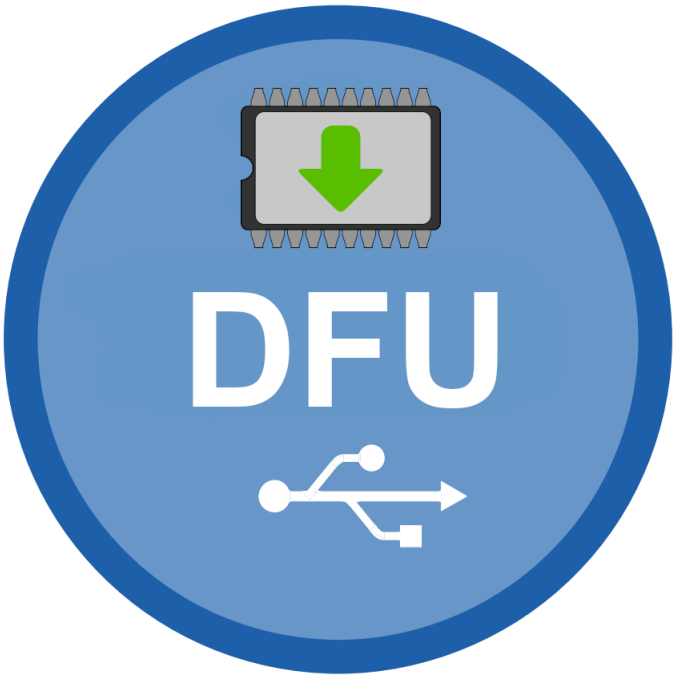
Parameter	Description
`FrameIndex`	1-based index of the frame resolution.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 20

Device Firmware Upgrade (DFU)

This chapter gives a general overview of the DFU class and describes how to get the DFU component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

20.1 Overview

The Device Firmware Upgrade class (DFU) is a USB class protocol which can be used to download and upload firmware images to and from a device.

emUSB-Device-DFU comes as a complete package and contains the following:

- Generic USB handling
- USB DFU class implementation (version 1.1)
- Sample application showing how to work with DFU

DFU is supported on most operating systems by common tools like `dfu-util`, see dfu-util.sourceforge.net.

20.1.1 Using DFU on Windows

In order to get emUSB DFU running with the WinUSB driver, the function `USBD_DFU_SetMSDescInfo()` must be called in the target application.

Microsoft's Windows operating systems (starting with XP Service Pack 2) contain a generic driver called WinUSB.sys that is used to handle all communication to an emUSB-Device running a DFU interface. If such a device is connected to a Windows 8, 8.1, 10 or 11 PC for the first time, Windows will install the WinUSB driver automatically. For Windows versions less than Windows 8, Microsoft provides a driver for Windows Vista and Windows 7, but this needs to be installed manually. A driver installation tool including the mentioned driver is available in the `Windows\USB\Bulk\WinUSBInstall`. Windows XP users can use the driver package located under `Windows\USB\Bulk\WinUSB_USBBulk_XP`.

Additionally, the correct driver may not be loaded on Windows 7 systems because of an issue in the USB 3.0 stack of Windows 7, see *Issues on Windows 7* on page 728.

 © 2010-2026 SEGGER Microcontroller GmbH

20.2 Configuration

20.2.1 Dual configuration mode

Typically, a device that supports DFU has to provide two different configurations. It starts up in runtime mode with the DFU interface and other interfaces used for normal operation of the device. In this configuration the device does not allow download or upload of firmware files.

If the host sends a DFU detach request, the device has to reconfigure to DFU mode, usually providing only a single DFU interface. DFU mode may for example be implemented by running a bootloader built into the device. In order to switch to DFU mode after receiving the detach request from the host, the device has to shut down and de-initialize the USB stack and start it up again using the DFU configuration. Then firmware files can be downloaded.

This procedure is compliant with the USB Device Firmware Upgrade class specification. See sample application `USB_DFU_Start.c`.

20.2.2 Single configuration

However, some DFU-capable devices do not offer two configurations. Instead, they are always in runtime mode, but allow up- and download of firmware files without changing the configuration. The emUSB-Device DFU class also supports this behavior, which is called 'Mixed Mode', see sample application `USB_DFU_MixedMode_Start.c`.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3 Target API

Function	Description
API functions	
USBD_DFU_Add()	Adds a DFU class interface.
USBD_DFU_Add_RunTime()	Adds a DFU class interface for runtime mode only.
USBD_DFU_AddAlternateInterface()	Adds an alternative interface to the DFU configuration.
USBD_DFU_SetMSDescInfo()	Enables use of Microsoft OS Descriptors.
USBD_DFU_SetPollTimeout()	Set the poll timeout to be reported to the host on the next GET_STATUS setup request.
USBD_DFU_Ack()	Acknowledge download data received via the USBD_DFU_DOWNLOAD function.
USBD_DFU_SetError()	Signal an error to the host.
USBD_DFU_ManifestComplt()	Must be called by the application after the new firmware was installed successfully.
USBD_DFU_GetStatusReqCnt()	Return the number of times, the host has requested a status after calling one of the functions USBD_DFU_Ack(), USBD_DFU_ManifestComplt() or USBD_DFU_GetStatusReqCnt().
USBD_DFU_GetAlternateSetting()	Returns the alternate interface setting that was set by the host.
Data structures	
USB_DFU_INIT_DATA	Initialization data for the DFU interface.
Function prototypes	
USBD_DFU_DETACH_REQUEST	Callback function is called when the host requests a DETACH, prompting the device to enter DFU mode.
USBD_DFU_DOWNLOAD	Callback function to handle download data to the application that was received from the host.
USBD_DFU_UPLOAD	Callback function to get upload data to be transferred to the host.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.1 API functions

20.3.1.1 USBD_DFU_Add()

Description

Adds a DFU class interface.

Prototype

```
void USBD_DFU_Add(const USB_DFU_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to USB_DFU_INIT_DATA structure.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.1.2 USBD_DFU_Add_RunTime()

Description

Adds a DFU class interface for runtime mode only. Using this function results in a smaller footprint than `USBD_DFU_Add()`.

Prototype

```
void USBD_DFU_Add_RunTime(const USB_DFU_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
`pInitData`	Pointer to `USB_DFU_INIT_DATA` structure. The field 'Mode' must be set to `USB_DFU_MODE_RUNTIME`.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.1.3 USBD_DFU_AddAlternateInterface()

Description

Adds an alternative interface to the DFU configuration. Must be called after `USBD_DFU_Add()` and before `USBD_Start()`. This function must not be called in runtime only mode.

Prototype

```
void USBD_DFU_AddAlternateInterface(const char * pInterfaceName);
```

Parameters

Parameter	Description
`pInterfaceName`	Pointer to a string containing the name of the alternate interface. The pointer must remain valid during all USB operations (until `USBD_DeInit()` is called).

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.1.4 USBD_DFU_SetMSDescInfo()

Description

Enables use of Microsoft OS Descriptors. A USB DFU device providing these descriptors is detected by Windows to be handled by the generic WinUSB driver.

Prototype

```
void USBD_DFU_SetMSDescInfo(void);
```

Additional information

This function must be called after the call to the function `USBD_DFU_Add()` and before `USBD_Start()`.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.1.5 USBD_DFU_SetPollTimeout()

Description

Set the poll timeout to be reported to the host on the next GET_STATUS setup request.

Prototype

```
void USBD_DFU_SetPollTimeout(U32 PollTimeout);
```

Parameters

Parameter	Description
PollTimeout	Poll timeout in milliseconds.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.1.6 USBD_DFU_Ack()

Description

Acknowledge download data received via the `USBD_DFU_DOWNLOAD` function.

Prototype

```
void USBD_DFU_Ack(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.1.7 USBD_DFU_SetError()

Description

Signal an error to the host. Can be called by the application at any time. The device will respond with that error on the next 'Get Status' request from the host.

Prototype

```
void USBD_DFU_SetError(USB_DFU_ERROR_STATE Err);
```

Parameters

Parameter	Description
Err	Error code.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.1.8 USBD_DFU_ManifestComplt()

Description

Must be called by the application after the new firmware was installed successfully.

Prototype

```
void USBD_DFU_ManifestComplt(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.1.9 USBD_DFU_GetStatusReqCnt()

Description

Return the number of times, the host has requested a status after calling one of the functions `USBD_DFU_Ack()`, `USBD_DFU_ManifestComplt()` or `USBD_DFU_GetStatusReqCnt()`.

Prototype

```
unsigned USBD_DFU_GetStatusReqCnt(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.1.10 USBD_DFU_GetAlternateSetting()

Description

Returns the alternate interface setting that was set by the host.

Prototype

```
unsigned USBD_DFU_GetAlternateSetting(void);
```

Return value

- 0 - No alternate interface was selected by the host.
- 1..n - Number of alternate interface selected by the host.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.2 Data structures

20.3.2.1 USB_DFU_INIT_DATA

Description

Initialization data for the DFU interface.

Type definition

```
typedef struct {
  I8                          Mode;
  U8                          Attributes;
  U16                         DetachTimeout;
  U16                         TransferSize;
  U16                         Flags;
  const char                * pInterfaceName;
  USBD_DFU_DETACH_REQUEST   * pfDetachRequest;
  USBD_DFU_DOWNLOAD         * pfDownload;
  U8                        * pBuffer;
  USBD_DFU_UPLOAD           * pfUpload;
} USB_DFU_INIT_DATA;
```

Structure members

Member	Description
Mode	Operation mode of the DFU interface: USB_DFU_MODE_RUNTIME: The interface is in runtime mode only. Download of firmware data is not supported. USB_DFU_MODE_DFU: The interface is in DFU mode. USB_DFU_MODE_MIXED: The interface is in runtime mode but allows download of firmware data in this mode.
Attributes	Bit mask containing the DFU attributes. Combination of the USB_DFU_ATTR_... flags.
DetachTimeout	Time, in milliseconds, that the device will wait after receipt of the DFU_DETACH request.
TransferSize	Maximum number of bytes that the device can accept per control-write transaction.
Flags	RFU. Must be 0.
pInterfaceName	Name of the interface. Optional, may be NULL.
pfDetachRequest	Pointer to the callback function to request a detach. Used for Mode = USB_DFU_MODE_RUNTIME only.
pfDownload	Pointer to the callback function to receive download data. Used for Mode ≠ USB_DFU_MODE_RUNTIME only.
pBuffer	Pointer to a buffer to store download data. The size of the buffer must be 'TransferSize' bytes.
pfUpload	Pointer to the callback function to get upload data. Optional. Used for Mode ≠ USB_DFU_MODE_RUNTIME only.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.3 Function prototypes

20.3.3.1 USBD_DFU_DETACH_REQUEST

Description

Callback function is called when the host requests a DETACH, prompting the device to enter DFU mode. This function is executed in interrupt context. The detach and/or reinitialization must not be performed inside this function. Instead this function should only trigger a task to perform the required operation.

Type definition

```
typedef void USBD_DFU_DETACH_REQUEST(U16 Timeout);
```

Parameters

Parameter	Description
Timeout	Timeout provided by the host.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.3.2 USBD_DFU_DOWNLOAD

Description

Callback function to handle download data to the application that was received from the host. The function is called in interrupt context and should return as fast as possible. Especially flash programming must not be done within this function. If `NumBytes` ≥ 0, the application must respond either with a call to `USBD_DFU_Ack()` if the data could be processed successfully or by calling `USBD_DFU_SetError()` if an error occurred. These functions need not to be called from the `USBD_DFU_DOWNLOAD` function, but may be called later after processing the data. The host will wait for either `USBD_DFU_Ack()` or `USBD_DFU_SetError()` before starting another download.

Type definition

```
typedef void USBD_DFU_DOWNLOAD(int NumBytes,
                               U16 BlockNum);
```

Parameters

Parameter	Description
`NumBytes`	Number of bytes received from the host. The data is stored in the buffer provided by `USB_DFU_INIT_DATA.pBuffer`. A value of 0 indicates the end of the data to be downloaded. A negative value means that the host has aborted the download.
`BlockNum`	Block sequence number provided by the host.

 © 2010-2026 SEGGER Microcontroller GmbH

20.3.3.3 USBD_DFU_UPLOAD

Description

Callback function to get upload data to be transferred to the host. The function is called in interrupt context and should return as fast as possible.

Type definition

```
typedef int USBD_DFU_UPLOAD(      int   bStart,
                                  U16   BlockNum,
                                  U16   NumBytes,
                            const U8 ** ppData);
```

Parameters

Parameter	Description
bStart	1 = Start upload, 0 = continue upload.
BlockNum	Block sequence number provided by the host.
NumBytes	Number of bytes requested by the host.
ppData	out Pointer to the data to be transferred to the host.

Return value

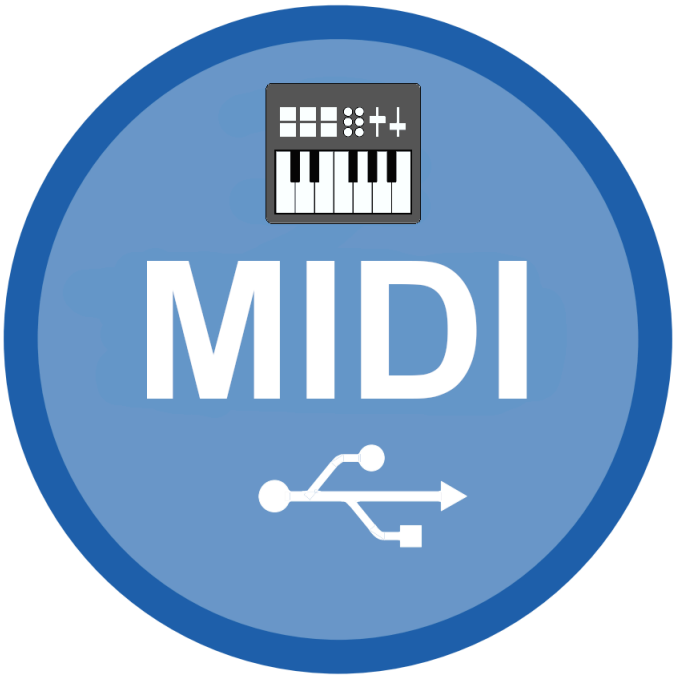
Size of the data provided by the function (in bytes). A value < NumBytes (including 0) indicates the last part of the data. A negative value indicates an error. In case of an error, the function should also call USBD_DFU_SetError().

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 21

Musical Instrument Digital Interface (MIDI)

This chapter gives a general overview of the Musical Instrument Digital Interface class and describes how to get the MIDI component running on the target.



 © 2010-2026 SEGGER Microcontroller GmbH

21.1 Overview

The USB MIDI device class is a subclass of the USB audio class. Despite being a subclass of the audio class, the protocol is almost entirely different. The MIDI class is able to transfer MIDI commands and MIDI data from a device to a host and vice versa.

MIDI is supported by most operating systems out of the box and the installation of additional drivers is not required.

emUSB-Device-MIDI comes as a complete package and contains the following:

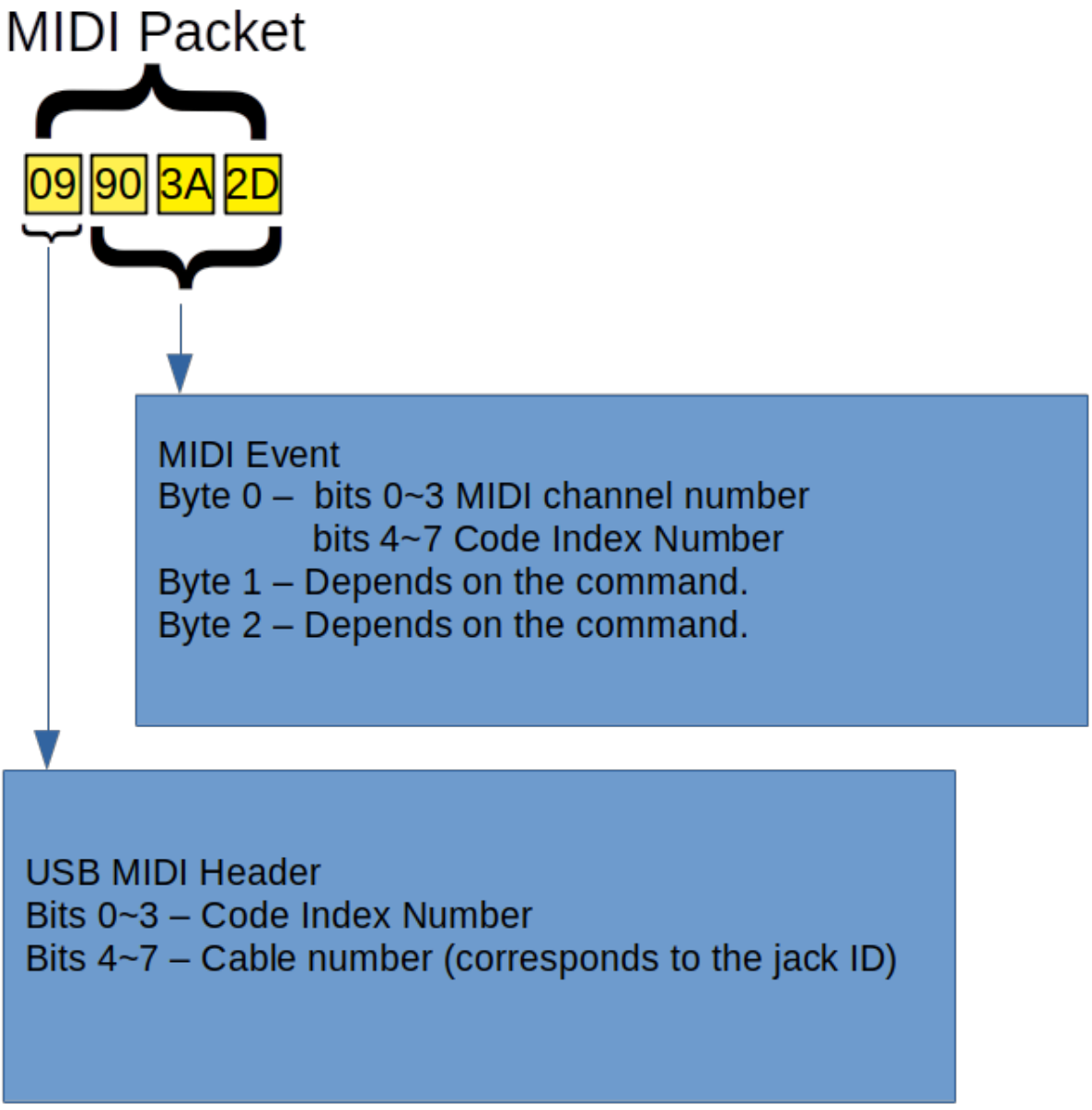
- Generic USB handling
- USB MIDI V1.0 device class implementation
- Sample application showing how to work with MIDI

 © 2010-2026 SEGGER Microcontroller GmbH

21.2 Introduction

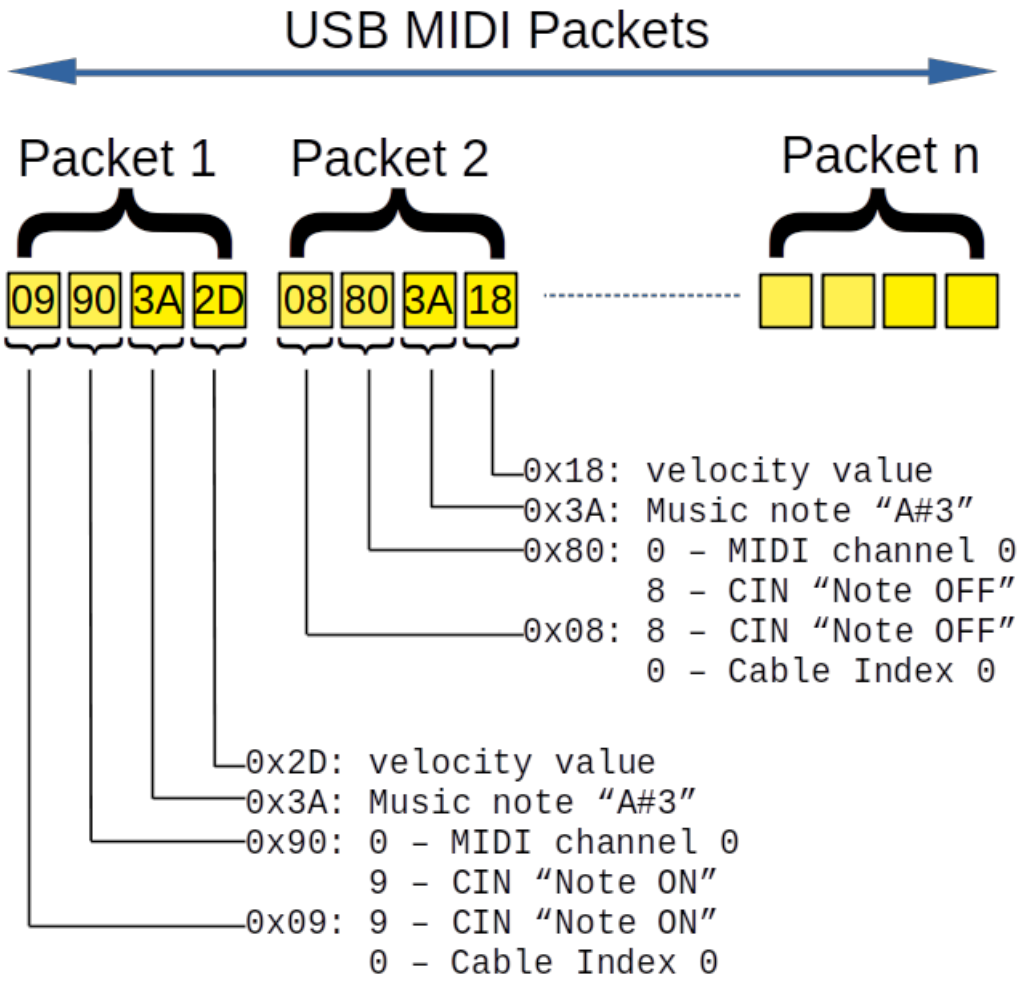
SEGGER's implementation of the MIDI class V1.0 is designed with minimal resource usage in mind, especially targeted to embedded devices. The implementation supports an arbitrary jack configuration as well as writing USB MIDI packets from a MIDI stream and receiving USB MIDI from a host.

The following graphic describes the basic structure of a USB MIDI packet.



 © 2010-2026 SEGGER Microcontroller GmbH

The MIDI commands “note ON” and “note OFF” can be seen in their decoded form below.



 © 2010-2026 SEGGER Microcontroller GmbH

21.3 Configuration

21.3.1 Initial configuration

To get emUSB-Device-MIDI up and running as well as to do an initial test, the configuration as delivered with the sample application should not be modified.

21.3.2 Final configuration

The configuration must only be modified when emUSB-Device is integrated in your final product. Refer to section *emUSB-Device Configuration* on page 53 for detailed information about the generic information functions which have to be adapted.

21.3.3 Testing MIDI on different operating systems

Linux

Install the third-party amidi command-line utility.

- Connect the device and call amidi -l. You should see a list of connected MIDI devices:

```
Dir Device    Name
IO  hw:2,0,0  MIDI device MIDI 1
```

- Using the device name, you can either send data to the device or receive data from the device.

Sending:

```
amidi -p hw:2,0,0 -S '90 4E 30 80 4E 30'  // 90 - Note on
                                          // 4E - Note "F#5"
                                          // 30 - Velocity value
                                          // 80 - Note off
                                          // 4E - Note "F#5"
                                          // 30 - Velocity value
```

Receiving:

```
amidi -p hw:2,0,0 -d
```

Windows

On Windows a third-party utility such as MIDI-OX can be used to monitor MIDI events.

Mac

On macOS a third-party utility such as Snoize MIDI Monitor can be used to monitor MIDI events.

 © 2010-2026 SEGGER Microcontroller GmbH

21.4 Target API

Function	Description
API functions	
USBD_MIDI_Init()	Initialize the MIDI component.
USBD_MIDI_Add()	Adds a MIDI class interface to the USB stack.
USBD_MIDI_ReceivePackets()	Receives USB MIDI packets from the host.
USBD_MIDI_GetNumPacketsInBuffer()	Returns the number of MIDI packets that are available in the internal OUT endpoint buffer.
USBD_MIDI_ConvertPackets()	Converts USB MIDI packets to pure MIDI commands by stripping the USB header.
USBD_MIDI_WritePackets()	Writes USB MIDI packets to the host.
USBD_MIDI_WriteStream()	Sends MIDI data to the USB host.
Data structures	
USBD_MIDI_INIT_DATA	Initialization structure that is needed when adding a MIDI interface to emUSB-Device.
USBD_MIDI_JACK	Structure describing a MIDI IN or OUT jack.
USBD_MIDI_PACKET	Structure describing a MIDI packet.

 © 2010-2026 SEGGER Microcontroller GmbH

21.4.1 API functions

21.4.1.1 USBD_MIDI_Init()

Description

Initialize the MIDI component.

Prototype

```
void USBD_MIDI_Init(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

21.4.1.2 USBD_MIDI_Add()

Description

Adds a MIDI class interface to the USB stack.

Prototype

```
USBD_MIDI_HANDLE USBD_MIDI_Add(const USBD_MIDI_INIT_DATA * pInitData);
```

Parameters

Parameter	Description
pInitData	Pointer to USBD_MIDI_INIT_DATA structure.

Return value

Handle to a valid MIDI instance. The handle of the first MIDI instance is always 0.

 © 2010-2026 SEGGER Microcontroller GmbH

21.4.1.3 USBD_MIDI_ReceivePackets()

Description

Receives USB MIDI packets from the host. The function blocks until any data has been received or a timeout occurs (if Timeout ≥ 0). In case of a timeout, the read transfer is aborted.

Prototype

```
int USBD_MIDI_ReceivePackets(USBD_MIDI_HANDLE   hInst,
                             USBD_MIDI_PACKET * paPacket,
                             unsigned           NumPackets,
                             int                Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid MIDI instance, returned by USBD_MIDI_Add().
paPacket	Pointer to an array of USBD_MIDI_PACKET structures.
NumPackets	Number of packets inside the paPacket array.
Timeout	Timeout in milliseconds. 0 means infinite. If Timeout is -1, the function never blocks and only reads data from the internal endpoint buffer.

Return value

> 0	Number of MIDI packets read.
= 0	A timeout occurred (if Timeout > 0), no data in buffer (if Timeout < 0) or the target was disconnected during the function call and no data was read so far.
< 0	Error occurred.

Additional information

This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return USB_STATUS_ERROR.

Periodically calling this function with timeout = -1 can be used to poll for data.

If the USB stack receives a data packet from the host containing more bytes than requested, the remaining bytes are stored into the internal buffer of the endpoint, that was provided via USBD_AddEP(). This data can be retrieved by a later call to USBD_MIDI_ReceivePackets(). See also USBD_MIDI_GetNumPacketsInBuffer().

 © 2010-2026 SEGGER Microcontroller GmbH

21.4.1.4 USBD_MIDI_GetNumPacketsInBuffer()

Description

Returns the number of MIDI packets that are available in the internal OUT endpoint buffer.

Prototype

```
unsigned USBD_MIDI_GetNumPacketsInBuffer(USBD_MIDI_HANDLE hInst);
```

Parameters

Parameter	Description
hInst	Handle to a valid MIDI instance, returned by USBD_MIDI_Add().

Return value

Number of packets that are available in the internal OUT endpoint buffer.

Additional information

If the host is sending more data than your target application has requested, the remaining data will be stored in an internal buffer. This function shows how many bytes are available in this buffer.

The number of packets returned by this function can be read using USBD_MIDI_ReceivePackets().

 © 2010-2026 SEGGER Microcontroller GmbH

21.4.1.5 USBD_MIDI_ConvertPackets()

Description

Converts USB MIDI packets to pure MIDI commands by stripping the USB header. USB MIDI packets are usually provided through the use of the USBD_MIDI_ReceivePackets() function.

Prototype

```
int USBD_MIDI_ConvertPackets(const USBD_MIDI_PACKET * paPacket,
                             unsigned                 NumPackets,
                             U8                     * pBuf);
```

Parameters

Parameter	Description
paPacket	Pointer to an array of USBD_MIDI_PACKET structures.
NumPackets	Number of packets inside the paPacket array.
pBuf	Buffer to write the MIDI commands into. The buffer must be 3 * NumPackets bytes large.

Return value

≥ 0 Number of MIDI packets converted.
< 0 Error occurred.

Additional information

The jack ID is lost in this conversion. When using multiple jacks the user should take care not to mix packets addressed to different jacks.

 © 2010-2026 SEGGER Microcontroller GmbH

21.4.1.6 USBD_MIDI_WritePackets()

Description

Writes USB MIDI packets to the host. Unlike with USBD_MIDI_WriteStream() the user must set the correct values for the USB MIDI header (CIN and jack ID) as these are not automatically filled in.

Prototype

```
int USBD_MIDI_WritePackets(      USBD_MIDI_HANDLE   hInst,
                           const USBD_MIDI_PACKET * paPacket,
                                 unsigned           NumPackets,
                                 int                Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid MIDI instance, returned by USBD_MIDI_Add().
paPacket	Pointer to an array of USBD_MIDI_PACKET structures. The user must fill all fields. For MIDI events which do not use all 3 MIDI bytes the user must fill the unused bytes with zeroes.
NumPackets	Number of packets inside the paPacket array.
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

> 0 Number of written USB MIDI packets.
= 0 A timeout occurred (if Timeout > 0).
< 0 Error occurred.

Additional information

This function also returns when the target is disconnected from the host or when a USB reset occurred during the function call, it will then return USB_STATUS_ERROR.

 © 2010-2026 SEGGER Microcontroller GmbH

21.4.1.7 USBD_MIDI_WriteStream()

Description

Sends MIDI data to the USB host. This function accepts a stream of MIDI commands and automatically adds the necessary USB MIDI header byte. Depending on the Timeout parameter, the function may block until NumBytes have been written or a timeout occurs.

Prototype

```
int USBD_MIDI_WriteStream(      USBD_MIDI_HANDLE   hInst,
                                U8                 JackID,
                          const U8               * pData,
                                unsigned           NumBytes,
                                int                Timeout);
```

Parameters

Parameter	Description
hInst	Handle to a valid MIDI instance, returned by USBD_MIDI_Add().
JackID	Jack ID to use.
pData	Data that should be written.
NumBytes	Number of bytes to write.
Timeout	Timeout in milliseconds. 0 means infinite.

Return value

= 0	Timeout has occurred and no data was written.
> 0 && < NumBytes	Number of bytes that have been written before a timeout occurred.
= NumBytes	Write transfer successfully completed.
< 0	Error occurred.

Additional information

This function also returns when the target is disconnected from host or when a USB reset occurred.

 © 2010-2026 SEGGER Microcontroller GmbH

21.4.2 Data structures

21.4.2.1 USBD_MIDI_INIT_DATA

Description

Initialization structure that is needed when adding a MIDI interface to emUSB-Device.

Type definition

```
typedef struct {
  U16                     Flags;
  U8                      EPIn;
  U8                      EPOut;
  const USBD_MIDI_JACK  * paJackList;
  unsigned                NumJacks;
} USBD_MIDI_INIT_DATA;
```

Structure members

Member	Description
Flags	Reserved for future use, must be 0.
EPIn	Bulk IN endpoint for sending data to the host.
EPOut	Bulk OUT endpoint for receiving data from the host.
paJackList	Pointer to an array containing all jacks for the MIDI interface.
NumJacks	Number of elements inside the paJackList array.

 © 2010-2026 SEGGER Microcontroller GmbH

21.4.2.2 USBD_MIDI_JACK

Description

Structure describing a MIDI IN or OUT jack.

Type definition

```
typedef struct {
  U8            JackType;
  U8            JackID;
  U8            JackDir;
  U8            NrInputPins;
  U8          * paSourceID;
  U8          * paSourcePin;
  const char  * pJackName;
} USBD_MIDI_JACK;
```

Structure members

Member	Description
JackType	USB_AUDIO_MIDI_EMBEDDED_JACK or USB_AUDIO_MIDI_EXTERNAL_JACK
JackID	Unique ID for the jack. Must not be zero.
JackDir	Jack direction, USB_AUDIO_MIDI_IN_JACK or USB_AUDIO_MIDI_OUT_JACK
NrInputPins	For IN jacks - set to zero. For OUT jacks - number of input pins for this MIDI OUT jack.
paSourceID	Only for OUT jacks. Pointer to an array containing the IDs of the entities to which the pin of this MIDI OUT Jack is connected.
paSourcePin	Only for OUT jacks. Pointer to an array containing the output pin numbers of the entities to which the input pins of this MIDI OUT Jack are connected.
pJackName	String describing the jack. Can be NULL.

 © 2010-2026 SEGGER Microcontroller GmbH

21.4.2.3 USBD_MIDI_PACKET

Description

Structure describing a MIDI packet.

Type definition

```
typedef struct {
  U8   CableNumber_and_CIN;
  U8   MIDI_0;
  U8   MIDI_1;
  U8   MIDI_2;
} USBD_MIDI_PACKET;
```

Structure members

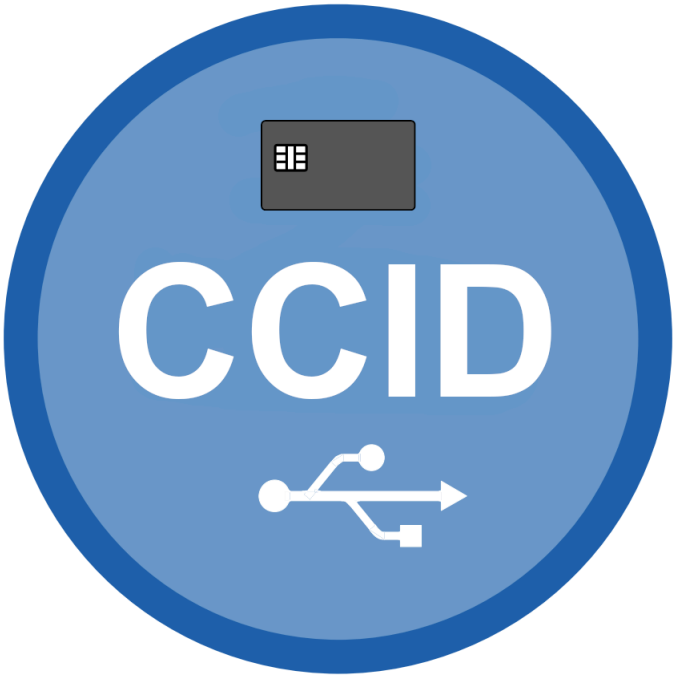
Member	Description
CableNumber_and_CIN	b0-b3 - Code Index Number (CIN). b4-b7 - Cable Number (embedded MIDI Jack ID).
MIDI_0	First MIDI byte: b0-b3 - For CIN < 0xF: MIDI channel number. b4-b7 - Code Index Number (same as in b0-b3 of CableNumber_and_CIN).
MIDI_1	Second MIDI byte - Content depends on CIN.
MIDI_2	Third MIDI byte - Content depends on CIN.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 22

Smart Card Device Class (CCID)

This chapter gives a general overview of the CCID class.



 © 2010-2026 SEGGER Microcontroller GmbH

22.1 Overview

The Smart Card Device Class (CCID) allows the implementation of CCID-compatible smart card readers. The Integrated Circuit(s) Cards Interface is an abstract USB class protocol defined by the USB Implementers Forum.

The emUSBD CCID class only handles the transport of CCID commands via USB. Processing of the smart card commands, including physical access to a smart card (if any), has to be done by the application.

A typical application will contain a loop that

- Reads a CCID command from the host using `USBD_CCID_ReceiveCmd()`.
- Processes the command depending on the message type and parameters.
- Sends an answer back to the host using one of the `USBD_CCID_Send…()` functions.

emUSB-Device CCID comes as a complete package and contains the following:

- Generic USB handling
- USB CCID class implementation (version 1.1)
- Sample application showing how to implement a simple card reader

 © 2010-2026 SEGGER Microcontroller GmbH

22.2 Target API

Function	Description
API functions	
USBD_CCID_Init()	Initialize the CCID component.
USBD_CCID_Add()	Adds interface for USB-CCID communication to emUSB-Device.
USBD_CCID_ReceiveCmd()	Receive a CCID command from the host.
USBD_CCID_SendStatus()	Send a response to a CCID command with message types USB_CCID_MSG_ICC_POWER_OFF, USB_CCID_MSG_GET_SLOT_STATUS, USB_CCID_MSG_ICC_CLOCK, USB_CCID_MSG_T0APDU or USB_CCID_MSG_MECHANICAL.
USBD_CCID_SendDataBlock()	Send a response to a CCID command with message types USB_CCID_MSG_XFR_BLOCK, USB_CCID_MSG_ICC_POWER_ON, or USB_CCID_MSG_SECURE.
USBD_CCID_SendEscape()	Send a response to a CCID command with message type USB_CCID_MSG_ESCAPE_CMD.
USBD_CCID_SendParameters()	Send a response to a CCID command with message type USB_CCID_MSG_SET_RATE_AND_CLOCK.
USBD_CCID_SendDataRateAndClockFrequency()	Send a response to a CCID command with message type USB_CCID_MSG_SET_RATE_AND_CLOCK.
USBD_CCID_NotifySlotState()	Send a notification about a new slot state to the host (via interrupt EP).
USBD_CCID_NotifyHwError()	Send a notification about a hardware error to the host (via interrupt EP).
Data structures	
USB_CCID_INIT_DATA	Initialization structure that is needed when adding a CCID interface to emUSB-Device.
USB_CCID_PROPERTIES	Declares all properties of a CCID device.
USB_CCID_CMD	Contains information about a CCID command sent from the host.
USB_CCID_PROTOCOL_DATA_T0	Protocol parameters for T=0 protocol.
USB_CCID_PROTOCOL_DATA_T1	Protocol parameters for T=1 protocol.
Function prototypes	
USBD_CCID_ABORT_CB	Callback function to forward a CCID abort request from the host to the application.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1 API functions

22.2.1.1 USBD_CCID_Init()

Description

Initialize the CCID component.

Prototype

```
void USBD_CCID_Init(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1.2 USBD_CCID_Add()

Description

Adds interface for USB-CCID communication to emUSB-Device.

Prototype

```
void USBD_CCID_Add(const USB_CCID_INIT_DATA  * pInitData,
                   const USB_CCID_PROPERTIES * pProperties);
```

Parameters

Parameter	Description
pInitData	Pointer to a USB_CCID_INIT_DATA structure.
pProperties	Pointer to a USB_CCID_PROPERTIES containing all properties of the CCID device. The pointer must remain valid during all CCID operations.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1.3 USBD_CCID_ReceiveCmd()

Description

Receive a CCID command from the host.

Prototype

```
int USBD_CCID_ReceiveCmd(USB_CCID_CMD * pCmd,
                         unsigned       BuffSize,
                         U8           * pBuff,
                         unsigned       Timeout);
```

Parameters

Parameter	Description
pCmd	Pointer to a USB_CCID_CMD structure that will be filled by the function with information about the CCID command received.
BuffSize	Size of the buffer pointed to by pBuff. If a CCID command contains more data bytes than 'BuffSize', then only 'BuffSize' bytes are copied to the buffer and the remaining bytes are discarded.
pBuff	Buffer to receive the command data (abData).
Timeout	Timeout in ms to wait for a CCID command from the host.

Return value

> 0 Success.
= 0 A timeout has occurred.
< 0 An error occurred.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1.4 USBD_CCID_SendStatus()

Description

Send a response to a CCID command with message types USB_CCID_MSG_ICC_POWER_OFF, USB_CCID_MSG_GET_SLOT_STATUS, USB_CCID_MSG_ICC_CLOCK, USB_CCID_MSG_T0APDU or USB_CCID_MSG_MECHANICAL.

Prototype

```
int USBD_CCID_SendStatus(const USB_CCID_CMD * pCmd,
                         U8                   Status,
                         U8                   Error,
                         U8                   ClockStatus);
```

Parameters

Parameter	Description
pCmd	Pointer to a USB_CCID_CMD structure returned from a call to USBD_CCID_ReceiveCmd().
Status	Status code. One of the USB_CCID_STATUS_CMD_.. macros or'ed together with one of the USB_CCID_STATUS_ICC_.. macros.
Error	Error code, if Status bit USB_CCID_STATUS_CMD_FAIL is set.
ClockStatus	• 0: Clock running. • 1: Clock stopped in state L. • 2: Clock stopped in state H. • 3: Clock stopped in an unknown state.

Return value

> 0	Success.
= 0	A timeout has occurred.
< 0	An error occurred.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1.5 USBD_CCID_SendDataBlock()

Description

Send a response to a CCID command with message types USB_CCID_MSG_XFR_BLOCK, USB_CCID_MSG_ICC_POWER_ON, or USB_CCID_MSG_SECURE.

Prototype

```
int USBD_CCID_SendDataBlock(const USB_CCID_CMD * pCmd,
                                  U8             Status,
                                  U8             Error,
                                  U8             ChainParameter,
                                  unsigned       DataLen,
                            const U8           * pData);
```

Parameters

Parameter	Description
pCmd	Pointer to a USB_CCID_CMD structure returned from a call to USBD_CCID_ReceiveCmd().
Status	Status code. One of the USB_CCID_STATUS_CMD_.. macros or'ed together with one of the USB_CCID_STATUS_ICC_.. macros.
Error	Error code, if Status bit USB_CCID_STATUS_CMD_FAIL is set.
ChainParameter	For extended APDU level, indicates if the response is complete, to be continued or if the command APDU can continue.
DataLen	Number of data bytes to be returned to the host.
pData	Pointer to the data to be returned to the host (abData).

Return value

> 0	Success.
= 0	A timeout has occurred.
< 0	An error occurred.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1.6 USBD_CCID_SendEscape()

Description

Send a response to a CCID command with message type USB_CCID_MSG_ESCAPE_CMD.

Prototype

```
int USBD_CCID_SendEscape(const USB_CCID_CMD * pCmd,
                               U8             Status,
                               U8             Error,
                               unsigned       DataLen,
                         const U8           * pData);
```

Parameters

Parameter	Description
pCmd	Pointer to a USB_CCID_CMD structure returned from a call to USBD_CCID_ReceiveCmd().
Status	Status code. One of the USB_CCID_STATUS_CMD_.. macros or'ed together with one of the USB_CCID_STATUS_ICC_.. macros.
Error	Error code, if Status bit USB_CCID_STATUS_CMD_FAIL is set.
DataLen	Number of data bytes to be returned to the host.
pData	Pointer to the data to be returned to the host (abData).

Return value

> 0	Success.
= 0	A timeout has occurred.
< 0	An error occurred.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1.7 USBD_CCID_SendParameters()

Description

Send a response to a CCID command with message type USB_CCID_MSG_SET_RATE_AND_CLOCK.

Prototype

```
int USBD_CCID_SendParameters(const USB_CCID_CMD * pCmd,
                                   U8             Status,
                                   U8             Error,
                                   U8             ProtocolNum,
                             const void         * pProtocolData);
```

Parameters

Parameter	Description
pCmd	Pointer to a USB_CCID_CMD structure returned from a call to USBD_CCID_ReceiveCmd().
Status	Status code. One of the USB_CCID_STATUS_CMD_.. macros or'ed together with one of the USB_CCID_STATUS_ICC_.. macros.
Error	Error code, if Status bit USB_CCID_STATUS_CMD_FAIL is set.
ProtocolNum	• 0: Protocol data for T=0 • 1: Protocol data for T=1
pProtocolData	Pointer to the protocol data structure (USB_CCID_PROTOCOL_DATA_T0 or USB_CCID_PROTOCOL_DATA_T1)

Return value

> 0	Success.
= 0	A timeout has occurred.
< 0	An error occurred.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1.8 USBD_CCID_SendDataRateAndClockFrequency()

Description

Send a response to a CCID command with message type USB_CCID_MSG_SET_RATE_AND_CLOCK.

Prototype

```
int USBD_CCID_SendDataRateAndClockFrequency(const USB_CCID_CMD * pCmd,
                                            U8              Status,
                                            U8              Error,
                                            U32             ClockFrequency,
                                            U32             DataRate);
```

Parameters

Parameter	Description
pCmd	Pointer to a USB_CCID_CMD structure returned from a call to USBD_CCID_ReceiveCmd().
Status	Status code. One of the USB_CCID_STATUS_CMD_.. macros or'ed together with one of the USB_CCID_STATUS_ICC_.. macros.
Error	Error code, if Status bit USB_CCID_STATUS_CMD_FAIL is set.
ClockFrequency	Current setting of the ICC clock frequency in KHz.
DataRate	Current setting of the ICC data rate in bps.

Return value

> 0 Success.
= 0 A timeout has occurred.
< 0 An error occurred.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1.9 USBD_CCID_NotifySlotState()

Description

Send a notification about a new slot state to the host (via interrupt EP).

Prototype

```
void USBD_CCID_NotifySlotState(unsigned Slot,
                               unsigned State);
```

Parameters

Parameter	Description
Slot	Slot index (counting from 0).
State	New slot state. • 0: No ICC present. • 1: ICC Present.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.1.10 USBD_CCID_NotifyHwError()

Description

Send a notification about a hardware error to the host (via interrupt EP).

Prototype

```
void USBD_CCID_NotifyHwError(unsigned Slot,
                             U8       SeqNum,
                             U8       ErrorCode);
```

Parameters

Parameter	Description
Slot	Slot index (counting from 0).
SeqNum	Sequence number of bulk out command when the hardware error occurred.
ErrorCode	Error code.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.2 Data structures

22.2.2.1 USB_CCID_INIT_DATA

Description

Initialization structure that is needed when adding a CCID interface to emUSB-Device.

Type definition

```
typedef struct {
  U8                    EPIn;
  U8                    EPOut;
  U8                    EPInt;
  U8                  * pBuff;
  USBD_CCID_ABORT_CB  * pfAbort;
} USB_CCID_INIT_DATA;
```

Structure members

Member	Description
EPIn	Endpoint for sending data to the host.
EPOut	Endpoint for receiving data from the host.
EPInt	Endpoint for sending notification to the host. Optional, may be 0.
pBuff	Pointer to endpoint buffer for EPIn. Buffer should be able to hold one USB packet.
pfAbort	Callback function to signal an abort by the host. Optional, may be NULL.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.2.2 USB_CCID_PROPERTIES

Description

Declares all properties of a CCID device.

Type definition

```
typedef struct {
  U16           Flags;
  U8            NumSlots;
  U8            VoltageSupport;
  U8            Protocols;
  U8            NumClocks;
  U8            DefaultClockIdx;
  U8            NumDataRates;
  U8            DefaultDataRateIdx;
  const U32   * pClocks;
  const U32   * pDataRates;
  U32           MaxIFSD;
  U32           Features;
  U32           MaxMessageLength;
  U8            ClassGetResponse;
  U8            ClassEnvelope;
  U8            LCDLines;
  U8            LCDColumns;
  U8            PINSupport;
  const char  * pInterfaceName;
} USB_CCID_PROPERTIES;
```

Structure members

Member	Description
Flags	Reserved for future use, must be 0.
NumSlots	Number of card slots supported by the device (max. 4).
VoltageSupport	Supported voltages, see USB_CCID_VOLTAGE… macros.
Protocols	Supported protocols (T=0, T=1), see USB_CCID_PROTOCOL… macros.
NumClocks	Number of supported clock rates (number of entries in the table pointed to by pClocks). Must be ≥ 1.
DefaultClockIdx	Index of the default clock within the table pClocks.
NumDataRates	Number of supported data rates (number of entries in the table pointed to by pDataRates). May be 0 to indicate a range. In this case pDataRates must contain 2 entries (min, max) and DefaultDataRateIdx must be 0.
DefaultDataRateIdx	Index of the default data rate within the table pDataRates.
pClocks	Table of all supported clock rates in KHz in increasing order.
pDataRates	Table of all supported data rates in bps in increasing order.
MaxIFSD	Indicates the maximum IFSD supported by CCID for protocol T=1.
Features	This value indicates what intelligent features the CCID has. The value is a bitwise OR operation performed on the macros USB_CCID_FEATURE…
MaxMessageLength	Maximum CCID message length.
ClassGetResponse	Significant only for CCID that offers an APDU level for exchanges. Indicates the default class value used by the CCID when it sends a Get Response command to perform the

 © 2010-2026 SEGGER Microcontroller GmbH

Member	Description
	transportation of an APDU by T=0 protocol. Value `0xFF` indicates that the CCID echoes the class of the APDU.
`ClassEnvelope`	Significant only for CCID that offers an extended APDU level for exchanges. Indicates the default class value used by the CCID when it sends an Envelope command to perform the transportation of an extended APDU by T=0 protocol. Value `0xFF` indicates that the CCID echoes the class of the APDU.
`LCDLines`	Number of lines of the LCD display. 0 if no display supported.
`LCDColumns`	Number of characters per line of the LCD display. 0 if no display supported.
`PINSupport`	This value indicates what PIN support features the CCID has.
`pInterfaceName`	Name of the interface.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.2.3 USB_CCID_CMD

Description

Contains information about a CCID command sent from the host.

Type definition

```
typedef struct {
  U8         MessageType;
  U8         Slot;
  U8         SeqNum;
  U8         PowerSelect;
  U8         BWI;
  U8         LevelParameter;
  U8         ProtocolNumber;
  U8         ClockCommand;
  U8         ClassValid;
  U8         ClassGetResponse;
  U8         ClassEnvelope;
  U8         Function;
  unsigned   DataLen;
} USB_CCID_CMD;
```

Structure members

Member	Description
MessageType	Message type, see USB_CCID_MSG_... macros.
Slot	Card slot index (counting from 0).
SeqNum	Command sequence number.
PowerSelect	Only valid for message type USB_CCID_MSG_ICC_POWER_ON. Contains voltage that is applied to the ICC: • 0 - Automatic Voltage Selection • 1 - 5.0 volts • 2 - 3.0 volts • 3 - 1.8 volts.
BWI	Only valid for message types USB_CCID_MSG_XFR_BLOCK and USB_CCID_MSG_SECURE. Used to extend the CCIDs Block Waiting Timeout for this current transfer. The CCID shall timeout the block after "this number multiplied by the Block Waiting Time" has expired.
LevelParameter	Only valid for message types USB_CCID_MSG_XFR_BLOCK and USB_CCID_MSG_SECURE. Use changes depending on the exchange level reported by the class descriptor in dwFeatures field: • Character level: Size of expected data to be returned by the bulk-IN endpoint. • Extended APDU level: Indicates if APDU begins or ends in this command.
ProtocolNumber	Only valid for message types USB_CCID_MSG_SET_PARAMETERS. • 0: Structure for protocol T=0. • 1: Structure for protocol T=1.
ClockCommand	Only valid for message types USB_CCID_MSG_ICC_CLOCK. • 0: restarts Clock. • 1: Stops Clock.
ClassValid	Only valid for message types USB_CCID_MSG_T0APDU. Bit 0 = 1 indicates, that the field ClassGetResponse is valid. Bit 1 = 1 indicates, that the field ClassEnvelope is valid.

 © 2010-2026 SEGGER Microcontroller GmbH

Member	Description
ClassGetResponse	Only valid for message types USB_CCID_MSG_T0APDU. Value to force the class byte of the header in a Get Response command. Value = 0xFF indicates that the class byte of the Get Response command echoes the class byte of the APDU.
ClassEnvelope	Only valid for message types USB_CCID_MSG_T0APDU. Value to force the class byte of the header in a Envelope command. Value = 0xFF indicates that the class byte of the Envelope command echoes the class byte of the APDU.
Function	Only valid for message types USB_CCID_MSG_MECHANICAL. This value corresponds to the mechanical function being requested: • 1 - Accept Card. • 2 - Eject Card. • 3 - Capture Card. • 4 - Lock Card. • 5 - Unlock Card.
DataLen	Size of data sent with this command.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.2.4 USB_CCID_PROTOCOL_DATA_T0

Description

Protocol parameters for T=0 protocol.

Type definition

```
typedef struct {
  U8   FindexDindex;
  U8   TCCKST0;
  U8   GuardTimeT0;
  U8   WaitingIntegerT0;
  U8   ClockStop;
} USB_CCID_PROTOCOL_DATA_T0;
```

Structure members

Member	Description
FindexDindex	see USB CCID specification.
TCCKST0	see USB CCID specification.
GuardTimeT0	see USB CCID specification.
WaitingIntegerT0	see USB CCID specification.
ClockStop	see USB CCID specification.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.2.5 USB_CCID_PROTOCOL_DATA_T1

Description

Protocol parameters for T=1 protocol.

Type definition

```
typedef struct {
  U8   FindexDindex;
  U8   TCCKST1;
  U8   GuardTimeT1;
  U8   WaitingIntegerT1;
  U8   ClockStop;
  U8   IFSC;
  U8   NadValue;
} USB_CCID_PROTOCOL_DATA_T1;
```

Structure members

Member	Description
FindexDindex	see USB CCID specification.
TCCKST1	see USB CCID specification.
GuardTimeT1	see USB CCID specification.
WaitingIntegerT1	see USB CCID specification.
ClockStop	see USB CCID specification.
IFSC	see USB CCID specification.
NadValue	see USB CCID specification.

 © 2010-2026 SEGGER Microcontroller GmbH

22.2.3 Function prototypes

22.2.3.1 USBD_CCID_ABORT_CB

Description

Callback function to forward a CCID abort request from the host to the application. The function is called in interrupt context and should return as fast as possible.

Type definition

```
typedef void USBD_CCID_ABORT_CB(U8 Slot,
                                U8 SeqNum);
```

Parameters

Parameter	Description
Slot	Card slot index (counting from 0).
SeqNum	Sequence number of the command to abort.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 23

emUSB-Web add-on

This chapter gives a general overview of the emUSB-Web add-on and describes how to get the emUSB-Web add-on running on the target.

 © 2010-2026 SEGGER Microcontroller GmbH

23.1 Overview

The emUSB-Web add-on allows users to easily facilitate web-server access via USB.

emUSB-Web uses a PC tool to receive communication requests from a browser application and forward those via USB to an embedded device without using a TCP/IP stack.

emUSB-Web comes as a complete package and contains the following:

- emUSB-Web embedded sample application containing an abstraction for the emWeb web-server
- emUSB-Web PC tool (Linux / macOS / Windows)

 © 2010-2026 SEGGER Microcontroller GmbH

23.2 Requirements

In order to use emUSB-Web, the emUSB BULK component as well as the emWeb web-server are required.

The emUSB-Web PC application is required:

https://www.segger.com/products/connectivity/emusb-device/add-ons/emusb-web/

 © 2010-2026 SEGGER Microcontroller GmbH

23.3 Configuration

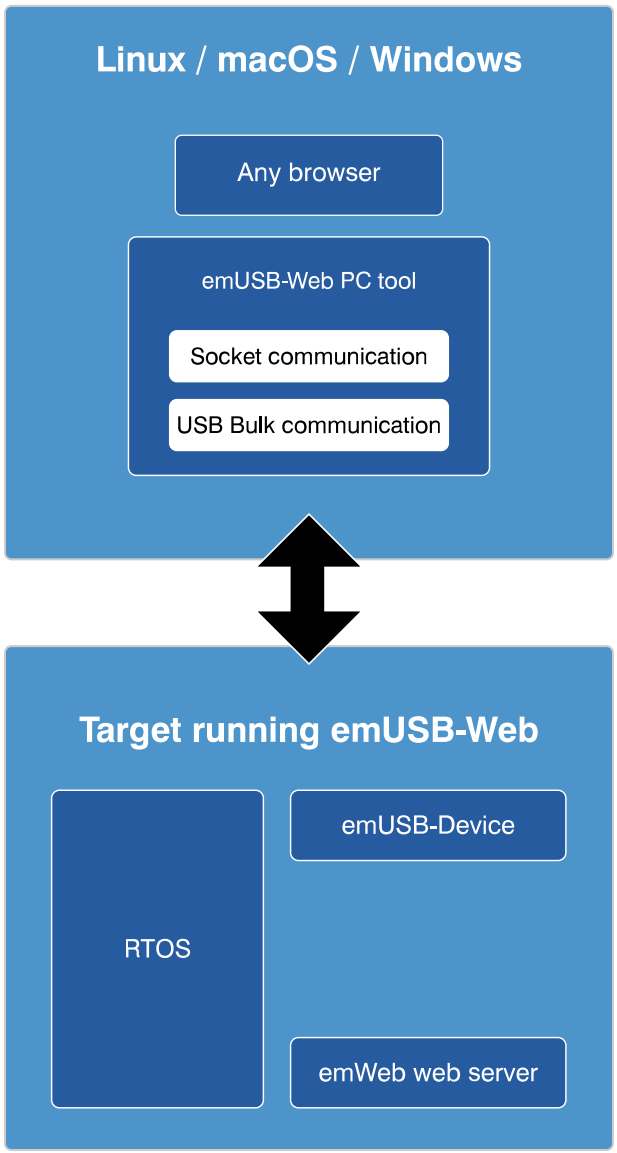
23.3.1 Initial configuration

The emUSB-Web PC application should be started. It will open the default browser and show a page instructing the user to connect a device running the emUSB-Web embedded counterpart.

The target device should be programmed with the emUSB-Web embedded application and connected to the PC.

As soon as a device running emUSB-Web is connected, the content of the web page is automatically replaced by the content provided by the embedded device.

23.3.1.1 emUSB-Web diagram



23.3.2 emUSB-Web operation in detail

The following chapter describes emUSB-Web internals and is relevant for users who wish to write their own PC application.

The USB web server consists of two parts: a PC application and an embedded application.

The PC application is responsible for opening a socket that a browser can connect to and forwarding any requests the browser sends to the embedded application via USB.

 © 2010-2026 SEGGER Microcontroller GmbH

The embedded application receives HTTP requests and processes them using the integrated web server. The response is sent back via USB, received by the PC application, and forwarded to the browser.

23.3.2.1 Device recognition

In order for the PC application to communicate with the embedded application, it must be able to identify which USB device it can communicate with. The embedded application must therefore provide a USB interface with the following characteristics:

- USB class ID: `0xFF` (Vendor specific)
- USB subclass ID: `0x57` (ASCII "W")
- USB protocol ID: `0x45` (ASCII "E")
- Interface string descriptor must contain the string "http"

23.3.2.2 emUSB-Web protocol

Communication between the PC application and the embedded application works via USB Bulk. The USB interface should contain one USB bulk IN endpoint and one USB bulk OUT endpoint.

After enumeration, the embedded application should start listening to commands from the PC application.

Commands from the PC application are preceded by an 8-byte-long header containing the following values:

- 4 Bytes "NumBytesDataDown" value indicating the number of data bytes following this header.
- 2 Bytes "Reserved1" - value reserved for future use.
- 2 Bytes "Reserved2" - value reserved for future use.

After reading the header, the embedded application should pass all following data to the embedded web server.

The reply from the web server must be sent back to the PC application. The reply must be preceded by an 8-byte-long header containing the following values:

- 4 Bytes "NumBytesDataUp" value indicating the number of data bytes following this header.
- 2 Bytes "RetVal" value. A value of zero indicates that more data will be sent by the web server for the current HTTP request. A value of 2 indicates that the current HTTP request will be completed with this transfer and that the PC can start the next HTTP request.
- 2 Bytes "Reserved" - value reserved for future use.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 24

Combining USB components (Multi-Interface)

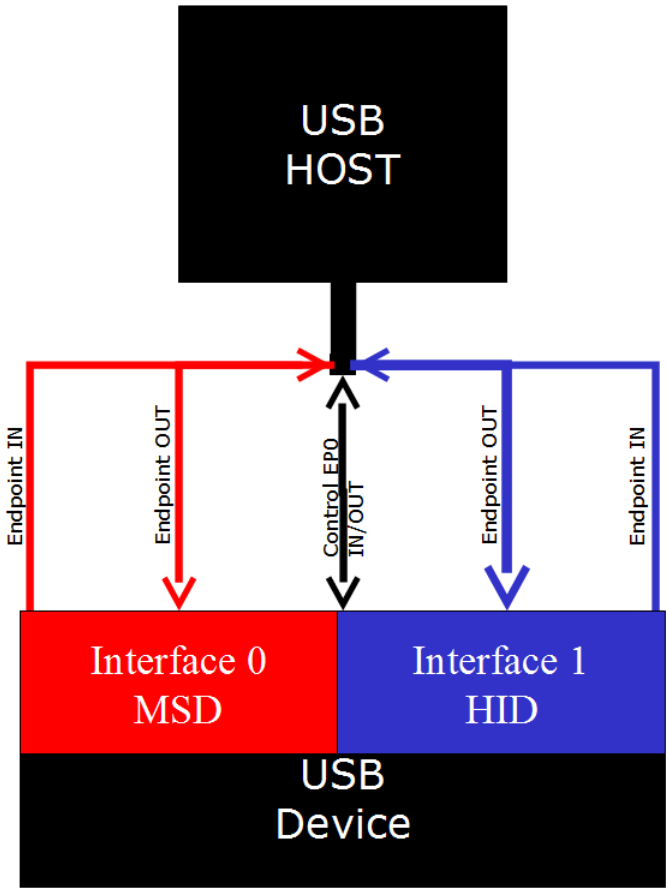
In some cases, it is necessary to combine different USB components in one device. This chapter will describe how to do this and which steps are necessary.

 © 2010-2026 SEGGER Microcontroller GmbH

24.1 Overview

The USB specification allows implementation of more than one component (function) in a single device. This is achieved by combining two or more components. These devices will be recognized by the USB host as a composite device and each component will be recognized as an independent device.

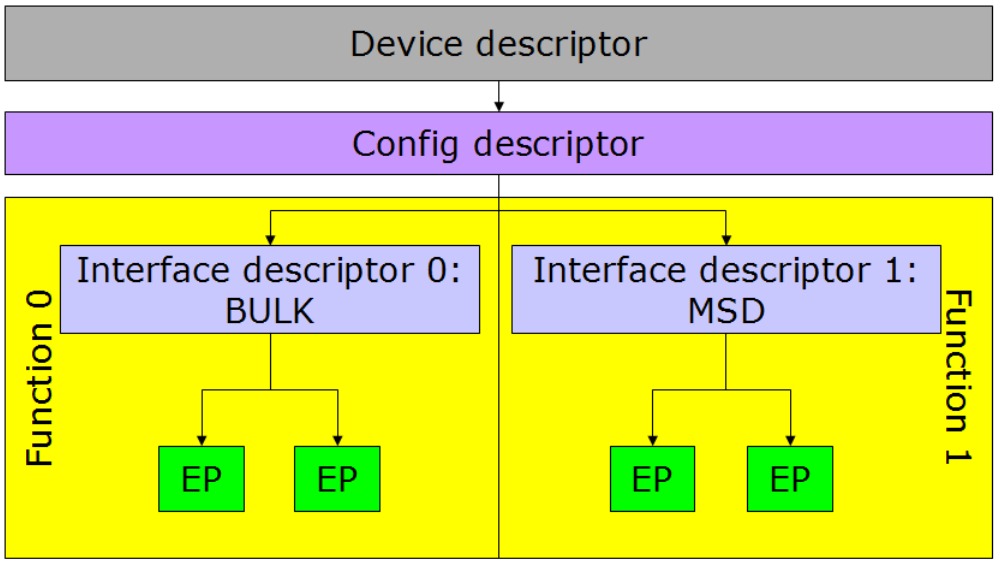
One device, for example a data logger, can have two components: This device can show log data files that were stored on a NAND flash through the MSD component. And the configuration of the data logger can be changed by using a BULK component, a CDC component, or even a HID component.



 © 2010-2026 SEGGER Microcontroller GmbH

24.1.1 Single interface device classes

Components can be combined because most USB device classes are based on one interface. This means that those components describe themselves at the interface descriptor level and thus make it easy to combine different or even the same device classes into one device. Such device classes are MSD, HID and generic bulk.



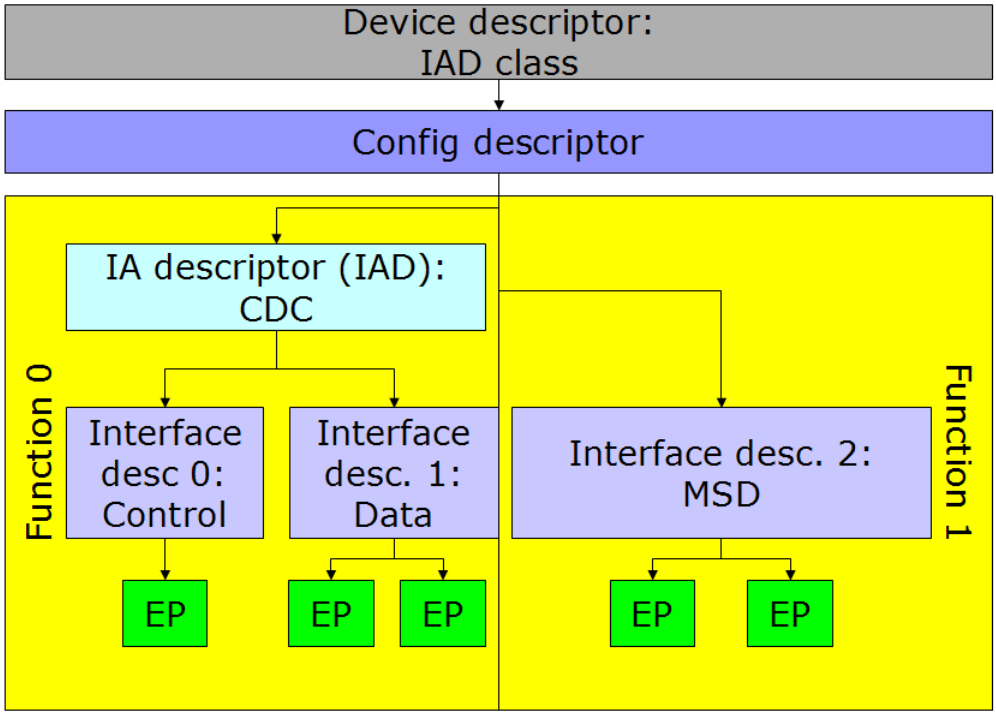
24.1.2 Multiple interface device classes

In contrast to the single interface classes, there are classes with multiple interfaces such as CDC, AUDIO, or VIDEO class. These classes define their class identifier in the device descriptor. All interface descriptors are recognized as part of the component that is defined in the device descriptor. This normally would prevent the combination of multiple interface device classes (for example, CDC) with any other component, but this can be avoided by using IAD.

24.1.3 IAD class

To remove the limitation described above, the USB protocol defines a descriptor type that allows the combination of single interface device classes with multiple interface device classes. This descriptor is called an Interface Association Descriptor (IAD). It decouples the multi-interface class from other interfaces.

 © 2010-2026 SEGGER Microcontroller GmbH



Since IAD is an extension to the original USB specification, it is not supported by all hosts, especially older host software. If IAD is not supported, the device may not be enumerated correctly.

Supported hosts

At the time of writing, IAD is supported by:

- Windows XP with Service Pack 2 and newer
- Linux Kernel 2.6.22 and higher
- macOS

 © 2010-2026 SEGGER Microcontroller GmbH

24.2 Configuration

In general, no configuration is required. By default, emUSB-Device supports up to four interfaces. If more interfaces are needed, the following macros must be modified:

Type	Macro	Default	Description
Numeric	USB_MAX_NUM_IF	4	Defines the maximum number of interfaces emUSB-Device shall handle.
Numeric	USB_MAX_NUM_IAD	3	Defines the maximum number of Interface Association Descriptors emUSB-Device shall handle.

 © 2010-2026 SEGGER Microcontroller GmbH

24.3 How to combine

Combining different single interface emUSB-Device components (Bulk, HID, MSD) is an easy step; all that needs to be done is calling the appropriate `USBD_xxx_Add()` function. For adding the CDC component, additional steps need to be taken. For detailed information, refer to *emUSB-Device component specific modification* on page and check the following sample.

Requirements

- Sufficient endpoints for all used device classes. Make sure that your USB device controller has enough endpoints available to handle all the interfaces that shall be integrated. The number of endpoints is limited by the USB controller hardware. This information is usually found in the specific MCU's reference manual.

Sample application

The following sample application uses embOS as the RTOS. This listing is taken from `USB_CompositeDevice_CDC_MSD.c`.

```
/*********************************************************************
*                   (c) SEGGER Microcontroller GmbH                  *
*                        The Embedded Experts                        *
*                           www.segger.com                           *
**********************************************************************

-------------------------- END-OF-HEADER -----------------------------

File    : USB_CompositeDevice_CDC_MSD.c
Purpose : Sample showing a USB device with multiple interfaces (CDC+MSD).
          This sample combines the functionality of USB_CDC_Echo.c
          and USB_MSD_FS_Start.c samples.

Additional information:
  Preparations:
    For CDC:
    On Windows 8.1 and below the "usbser" driver is not automatically
    assigned to the CDC-ACM device. To install the "usbser" driver
    see \Windows\USB\CDC . The device can be accessed via COM port
    emulation programs e.g. PuTTY.

    On Linux no drivers are needed, the device should show up as
    /dev/ttyACM0 or similar. "sudo screen /dev/ttyACM0 115200"
    can be used to access the device.

    On macOS no drivers are needed, the device should show up as
    /dev/tty.usbmodem13245678 or similar. The "screen" terminal
    program can be used to access the device.

    For MSD:
    The correct emFile configuration file has
    to be included in the project. Depending on the hardware
    it can be one of the following:
    * FS_ConfigRAMDisk_23k.c
    * FS_ConfigNAND_*.c
    * FS_ConfigMMC_CardMode_*.c
    * FS_ConfigNAND_*.c

  Expected behavior:
    For CDC:
    After connecting the USB cable the PC registers a new COM port appears.
    Terminal programs are able to open the COM port.
    Any data sent should be received back from the target.

    For MSD:
    A new MSD volume is recognized by the PC.
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
  Sample output:
    The target side does not produce terminal output.
*/

/*********************************************************************
*
*       #include section
*
**********************************************************************
*/
#include "USB.h"
#include "USB_CDC.h"
#include "BSP.h"
#include "USB_MSD.h"
#include "FS.h"
#include "RTOS.h"

/*********************************************************************
*
*       Static const data
*
**********************************************************************
*/
//
//  Information that is used during enumeration.
//
static const USB_DEVICE_INFO _DeviceInfo = {
  0x8765,         // VendorId
  0x1256,         // ProductId
  "Vendor",       // VendorName
  "MSD/CDC Composite device",  // ProductName
  "1234567890ABCDEF"           // SerialNumber
};
//
// String information used when inquiring the volume 0.
//
static const USB_MSD_LUN_INFO _Lun0Info = {
  "Vendor",     // MSD VendorName
  "MSD Volume", // MSD ProductName
  "1.00",       // MSD ProductVer
  "134657890"   // MSD SerialNo
};

/*********************************************************************
*
*       Static data
*
**********************************************************************
*/
// Data for MSD Task
static OS_STACKPTR int _aMSDStack[512]; /* Task stacks */
static OS_TASK _MSDTCB;               /* Task-control-blocks */

/*********************************************************************
*
*       Static code
*
**********************************************************************
*/

/*********************************************************************
*
*       _AddMSD
*
*  Function description
*    Add mass storage device to USB stack
*/
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
static void _AddMSD(void) {
  static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
  USB_MSD_INIT_DATA     InitData;
  USB_MSD_INST_DATA     InstData;

  InitData.EPIn  = USBD_AddEP(1, USB_TRANSFER_TYPE_BULK, 0, NULL, 0);
  InitData.EPOut = USBD_AddEP(0, USB_TRANSFER_TYPE_BULK, 0, _abOutBuffer, sizeof(_abOutBuffer));
  USBD_MSD_Add(&InitData);
  //
  // Add logical unit 0: RAM drive, using SDRAM
  //
  memset(&InstData, 0,  sizeof(InstData));
  InstData.pAPI                   = &USB_MSD_StorageByName;
  InstData.DriverData.pStart      = (void *)"";
  InstData.pLunInfo = &_Lun0Info;
  USBD_MSD_AddUnit(&InstData);
}
/*********************************************************************
*
*       _MSDTask
*
*  Function description
*    Add mass storage device to USB stack
*/
static void _MSDTask(void) {
  while (1) {
    while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED)) !
= USB_STAT_CONFIGURED) {
      USB_OS_Delay(50);
    }
    USBD_MSD_Task();
  }
}

/*********************************************************************
*
*       _OnLineCoding
*
*  Function description
*    Called whenever a "SetLineCoding" Packet has been received
*
*  Notes
*    (1) Context
*        This function is called directly from an ISR in most cases.
*/
static void _OnLineCoding(USB_CDC_LINE_CODING * pLineCoding) {
#if 0
  USBD_Logf_Application("DTERate=%u, CharFormat=%u, ParityType=%u, DataBits=%u
\n",
          pLineCoding->DTERate,
          pLineCoding->CharFormat,
          pLineCoding->ParityType,
          pLineCoding->DataBits);
#else
  BSP_USE_PARA(pLineCoding);
#endif
}

/*********************************************************************
*
*       _AddCDC
*
*  Function description
*    Add communication device class to USB stack
*/
static USB_CDC_HANDLE _AddCDC(void) {
  static U8 _abOutBuffer[USB_HS_BULK_MAX_PACKET_SIZE];
  USB_CDC_INIT_DATA     InitData;
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
  USB_CDC_HANDLE        hInst;

  InitData.EPIn  = USBD_AddEP(USB_DIR_IN,  USB_TRANSFER_TYPE_BULK, 0, NULL, 0);
  InitData.EPOut = USBD_AddEP(USB_DIR_OUT, USB_TRANSFER_TYPE_BULK, 0, _abOutBuffer, USB_HS_BULK_
  InitData.EPInt = USBD_AddEP(USB_DIR_IN,  USB_TRANSFER_TYPE_INT, 64,  NULL, 0);
  hInst = USBD_CDC_Add(&InitData);
  USBD_CDC_SetOnLineCoding(hInst, _OnLineCoding);
  return hInst;
}
/*********************************************************************
*
*       Public code
*
**********************************************************************
*/

/*********************************************************************
*
*       MainTask
*
* USB handling task.
*   Modify to implement the desired protocol
*/
#ifdef __cplusplus
extern "C" {     /* Make sure we have C-declarations in C++ programs */
#endif
void MainTask(void);
#ifdef __cplusplus
}
#endif
void MainTask(void) {
  USB_CDC_HANDLE hInstCDC;

  USBD_Init();
  USBD_EnableIAD();
  USBD_SetDeviceInfo(&_DeviceInfo);
  hInstCDC = _AddCDC();
  _AddMSD();
  USBD_Start();
  OS_CREATETASK(&_MSDTCB,  "MSDTask",  _MSDTask, 200, _aMSDStack);

  while (1) {
    char ac[64];
    int  NumBytesReceived;

    //
    // Wait for configuration
    //
    while ((USBD_GetState() & (USB_STAT_CONFIGURED | USB_STAT_SUSPENDED)) !
= USB_STAT_CONFIGURED) {
      BSP_ToggleLED(0);
      USB_OS_Delay(50);
    }
    BSP_SetLED(0);
    NumBytesReceived = USBD_CDC_Receive(hInstCDC, &ac[0], sizeof(ac), 0);
    if (NumBytesReceived > 0) {
      USBD_CDC_Write(hInstCDC, &ac[0], NumBytesReceived, 0);
    }
  }
}

/************************** end of file ******************************/
```

 © 2010-2026 SEGGER Microcontroller GmbH

24.4 emUSB-Device component-specific modification

There are different steps for each emUSB-Device component. The next section shows what needs to be done on both sides: device-side and host-side.

24.4.1 CDC component

24.4.1.1 Device side

In order to combine the CDC component with other components, the function `USBD_EnableIAD()` needs to be called, otherwise the device will not enumerate correctly. Refer to section How to combine on page 390 and check the listing of the sample application.

24.4.1.2 Host side

Due to a limitation of the internal CDC serial driver of Windows, a composite device with a CDC component and another device component(s) is only properly recognized by Windows XP SP3 and above. The Linux kernel supports IAD with version 2.6.22. For Windows before Windows 10, the .inf file needs to be modified. The provided `.inf` file:

```
;
;   Device installation file for
;   USB 2 COM port emulation
;
;
;
[Version]
Signature="$Windows NT$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%MFGNAME%
LayoutFile=layout.inf
DriverVer=03/26/2007,6.0.2600.1
CatalogFile=usbser.cat

[Manufacturer]
%MFGNAME%=CDCDevice,NT,NTamd64

[DestinationDirs]
DefaultDestDir = 12

[CDCDevice.NT]
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_1111&Mi_xx

[CDCDevice.NTamd64]
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_0234&Mi_xx
%DESCRIPTION%=DriverInstall,USB\VID_8765&PID_1111&Mi_xx

[DriverInstall.NT]
Include=mdmcpq.inf
CopyFiles=FakeModemCopyFileSection
AddReg=DriverInstall.NT.AddReg

[DriverInstall.NT.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,usbser.sys
HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"

[DriverInstall.NT.Services]
AddService=usbser, 0x00000002, DriverServiceInst

[DriverServiceInst]
DisplayName=%SERVICE%
ServiceType=1
StartType=3
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
ErrorControl=1
ServiceBinary=%12%\usbser.sys

[Strings]
MFGNAME = "Manufacturer"
DESCRIPTION = "USB CDC serial port emulation"
SERVICE = "USB CDC serial port emulation"
```

red - required modifications

Please add the red-colored text to your `.inf` file and change `xx` to the interface number of the CDC component.

The interface number is a zero-based index and is assigned by the emUSB-Device stack when calling the `USBD_CDC_Add()` function.

 © 2010-2026 SEGGER Microcontroller GmbH

24.5 Issues on Windows 7

When connecting a multi-interface device to Windows 7, sometimes a wrong driver is installed, causing the device not to work.

The device needs to be handled as a composite device, whereas Windows/third-party USB host software installs the driver for the first interface only.

This problem can be fixed manually using SEGGER's USB Composite Device Driver Fixer tool: https://www.segger.com/downloads/free-utilities/CompositeDeviceFixer.

24.5.1 Detailed description

The reason the driver was wrongly selected is that some USB 3.0 controller drivers include their own USB 3.0 stack, since the native USB host stack of Windows 7 cannot handle USB 3.0 devices (Microsoft introduced a native USB 3.0 stack only on Windows 8 and higher).

Normally, the third-party USB 3.0 stacks handle the enumeration and data transfers correctly. Unless it comes to MS OS descriptor handling. Almost all third-party stacks handle them incorrectly. Especially when it comes to devices which have multiple interfaces such as RNDIS+MSD or MTP+Bulk or RNDIS+WinuSB etc. or RNDIS+CDC. Even when IAD is used, it is wrongly passed. In normal cases Windows would initially report that a composite device is detected. Afterward, the individual interfaces are enumerated and the driver for each interface will be loaded. When MS OS Descriptors are used, the information is passed to the dedicated interfaces. Those third-party USB 3.0 stacks are NOT doing this. They pass this information to the device instead of the interface, which is wrong. Microsoft says in the MS OS Descriptor 2.0 Specification that the MS OS Descriptor information needs to be passed to the interfaces: [Microsoft OS 2.0 Descriptors Specification /April 2017 - Chapter Overview] "Scoping of MS OS descriptors With MS OS version 1.0 descriptors, Windows USB driver stack does not query for any MS OS descriptors if the device is a composite device, instead defers such queries to the USB Generic Parent Driver (Usbccgp.sys). The effect is that all MS OS descriptors are applied to specific composite functions, and none can be applied to the entire device itself."

Neither do those third-party USB host stacks implement a proper handling of the MS OS Descriptors nor do they pass the information properly to other drivers/stacks.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 25

Target OS Interface

This chapter describes the functions of the operating system abstraction layer.

 © 2010-2026 SEGGER Microcontroller GmbH

25.1 General information

emUSB-Device includes an OS abstraction layer which should make it possible to use an arbitrary operating system together with emUSB-Device. To adapt emUSB-Device to a new OS, one only has to map the functions listed below in section Interface function list to the native OS functions.

SEGGER took great care when designing this abstraction layer to make it easy to understand and to adapt to different operating systems.

25.1.1 Operating system support supplied with this release

emUSB-Device packages contain an abstraction layer for *embOS* (`USB_OS_embOSv5.c`). A kernel abstraction layer for using emUSB-Device without any RTOS (superloop) is also supplied (`USB_OS_None.c`).

Abstraction layers for the following operating systems are readily available:

- FreeRTOS
- µC/OS-II
- µC/OS-III
- CMSIS-RTX
- Keil-RTX
- ThreadX
- chibiOS
- CMX RTOS

Abstraction layers for other operating systems can be written upon request.

 © 2010-2026 SEGGER Microcontroller GmbH

25.2 Interface function list

Name	Description
API functions	
USB_OS_DeInit()	Frees all resources used by the OS layer.
USB_OS_Delay()	Delays for a given number of ms.
USB_OS_DecRI()	Leave a critical region for the USB stack: Decrements interrupt disable count and enable interrupts if counter reaches 0.
USB_OS_GetTickCnt()	Returns the current system time in milliseconds or system ticks.
USB_OS_IncDI()	Enter a critical region for the USB stack: Increments interrupt disable count and disables interrupts.
USB_OS_Init()	This function initializes all OS objects that are necessary.
USB_OS_Panic()	Is called if the stack encounters a fatal error.
USB_OS_Signal()	Wakes the task waiting for signal.
USB_OS_Wait()	Blocks the task until USB_OS_Signal() is called for a given transaction.
USB_OS_WaitTimed()	Blocks the task until USB_OS_Signal() is called for a given transaction or a timeout occurs.
USB_OS_MutexAlloc()	Allocates a new mutex to be used by USB_OS_MutexLock() / USB_OS_MutexUnlock() calls.
USB_OS_MutexFree()	Releases all mutexes allocated by USB_OS_MutexAlloc().
USB_OS_MutexLock()	This function locks a mutex object that was allocated by USB_OS_MutexAlloc().
USB_OS_MutexUnlock()	This function unlocks a mutex object that was allocated by USB_OS_MutexAlloc().

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.1 USB_OS_DeInit()

Description

Frees all resources used by the OS layer.

Prototype

```
void USB_OS_DeInit(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.2 USB_OS_Delay()

Description

Delays for a given number of ms.

Prototype

```
void USB_OS_Delay(int ms);
```

Parameters

Parameter	Description
ms	Number of ms.

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.3 USB_OS_DecRI()

Description

Leave a critical region for the USB stack: Decrements interrupt disable count and enable interrupts if counter reaches 0.

Prototype

```
void USB_OS_DecRI(void);
```

Additional information

The USB stack will perform nested calls to `USB_OS_IncDI()` and `USB_OS_DecRI()`. This function may be called from a task context or from within an interrupt. If called from an interrupt, it need not do anything.

An alternate implementation would be to

- enable the USB interrupts,
- unlock the mutex or semaphore locked in `USB_OS_IncDI()`

if the disable count reaches 0.

This may be more efficient, because interrupts of other peripherals can be serviced while inside a critical section of the USB stack.

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.4 USB_OS_GetTickCnt()

Description

Returns the current system time in milliseconds or system ticks.

Prototype

```
U32 USB_OS_GetTickCnt(void);
```

Return value

Current system time.

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.5 USB_OS_IncDI()

Description

Enter a critical region for the USB stack: Increments interrupt disable count and disables interrupts.

Prototype

```
void USB_OS_IncDI(void);
```

Additional information

The USB stack will perform nested calls to `USB_OS_IncDI()` and `USB_OS_DecRI()`. This function may be called from a task context or from within an interrupt. If called from an interrupt, it need not do anything.

An alternate implementation would be to

- perform a lock using a mutex or semaphore and
- disable the USB interrupts.

This may be more efficient, because interrupts of other peripherals can be serviced while inside a critical section of the USB stack.

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.6 USB_OS_Init()

Description

This function initializes all OS objects that are necessary.

Prototype

```
void USB_OS_Init(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.7 USB_OS_Signal()

Description

Wakes the task waiting for signal.

Prototype

```
void USB_OS_Signal(unsigned EPIndex,
                   unsigned TransactCnt);
```

Parameters

Parameter	Description
EPIndex	Endpoint index. Signaling must be independent for all end-points.
TransactCnt	Transaction counter. Specifies which transaction has been finished.

Additional information

This routine is typically called from within an interrupt service routine.

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.8 USB_OS_Wait()

Description

Blocks the task until `USB_OS_Signal()` is called for a given transaction.

Prototype

```
void USB_OS_Wait(unsigned EPIndex,
                 unsigned TransactCnt);
```

Parameters

Parameter	Description
`EPIndex`	Endpoint index. Signaling must be independent for all end-points.
`TransactCnt`	Transaction counter. Specifies the transaction to wait for.

Additional information

The function must ignore signaling transactions other than given in `TransactCnt`. If this transaction was signaled before this function was called, it must return immediately.

This routine is called from a task.

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.9 USB_OS_WaitTimed()

Description

Blocks the task until `USB_OS_Signal()` is called for a given transaction or a timeout occurs.

Prototype

```
int USB_OS_WaitTimed(unsigned EPIndex,
                     unsigned ms,
                     unsigned TransactCnt);
```

Parameters

Parameter	Description
`EPIndex`	Endpoint index. Signaling must be independent for all endpoints.
`ms`	Timeout time given in `ms`.
`TransactCnt`	Transaction counter. Specifies the transaction to wait for.

Return value

0	Task was signaled within the given timeout.
1	Timeout occurred.

Additional information

The function must ignore signaling transactions other than given in `TransactCnt`. If this transaction was signaled before this function was called, it must return immediately.

`USB_OS_WaitTimed()` is called from a task. This function is used by all available timed routines.

Alternatively this function may take the given timeout in units of system ticks of the underlying operating system instead of milliseconds. In this case all API functions that support a timeout parameter should also use system ticks for the timeout.

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.10 USB_OS_MutexAlloc()

Description

Allocates a new mutex to be used by `USB_OS_MutexLock()` / `USB_OS_MutexUnlock()` calls.

Prototype

```
int USB_OS_MutexAlloc(void);
```

Return value

≥ 0	Valid index to be used for `USB_OS_MutexLock()` / `USB_OS_MutexUnlock()`.
< 0	Error: No mutex available.

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.11 USB_OS_MutexFree()

Description

Releases all mutexes allocated by `USB_OS_MutexAlloc()`

Prototype

```
void USB_OS_MutexFree(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.12 USB_OS_MutexLock()

Description

This function locks a mutex object that was allocated by `USB_OS_MutexAlloc()`.

Prototype

```
void USB_OS_MutexLock(int Idx);
```

Parameters

Parameter	Description
`Idx`	Index of the mutex to be locked (from `USB_OS_MutexAlloc()`).

 © 2010-2026 SEGGER Microcontroller GmbH

25.2.13 USB_OS_MutexUnlock()

Description

This function unlocks a mutex object that was allocated by `USB_OS_MutexAlloc()`.

Prototype

```
void USB_OS_MutexUnlock(int Idx);
```

Parameters

Parameter	Description
`Idx`	Index of the mutex to be unlocked (from `USB_OS_MutexAlloc()`).

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 26

Target USB Driver

This chapter describes how to configure a USB driver for emUSB-Device in detail.

 © 2010-2026 SEGGER Microcontroller GmbH

26.1 General information

Purpose of the USB hardware interface

emUSB-Device does not contain any hardware dependencies. These are encapsulated through a hardware abstraction layer, which consists of the interface functions described in this chapter. All of these functions for a particular USB controller are typically located in a single file, the USB driver. Drivers for hardware which have already been tested with emUSB-Device are available.

Range of supported USB hardware

The interface has been designed in such a way that it should be possible to use the most common USB device controllers. This includes USB 1.1, USB 2.0 and USB 3.0 controllers.

26.1.1 Available USB drivers

An up-to-date list can always be found at:

https://www.segger.com/emusb-drivers.html

 © 2010-2026 SEGGER Microcontroller GmbH

26.2 Adding a driver to emUSB-Device

USBD_Init() initializes the internals of the USB stack and is always the first function which the USB application has to call. USBD_Init() will then call USBD_X_Config(). This function should be used to perform the following tasks:

- Perform device-specific hardware initialization if necessary.
- Add a USB driver to your project.
- Enable SuperSpeed by calling the function USBD_EnableSuperSpeed() if possible and desired.
- Assign a memory area to be used for endpoint buffers if required by the driver, see USBD_AssignMemory().
- Optionally install a HWAttach function.
- Install interrupt management functions.

You have to specify the USB device driver which should be used with emUSB-Device. For this, USBD_AddDriver() should be called in USBD_X_Config() with the identifier of the driver which is compatible with your hardware as a parameter. Refer to the header file USB.h or USB_HW_*.h for a list of all supported devices and their valid identifiers.

The _HWAttach() function should be used to perform hardware-specific actions which are not part of the USB controller logic (for example, enabling the peripheral clock for the USB port). This function is called from every device driver, but may not be present if your hardware does not need to perform such actions. A _HWAttach() function may be registered to the stack by calling USBD_SetAttachFunc() within USBD_X_Config().

Additionally, a function to enable the USB interrupt must be installed using the function USBD_SetISREnableFunc().

Modify USBD_X_Config(), _EnableISR() and if required, _HWAttach().

26.2.1 USBD_X_Config()

Description

Configure the USB stack.

Prototype

void USBD_X_Config(void)

Additional information

This function is always called from USBD_Init().

Example

```
/* Example excerpt from USB_Config_SAM7A3.c */
#define PID_USB          (27) //  USB Identifier
#define _AT91C_PIOA_BASE  (0xFFFFF400)
#define _AT91C_PIOB_BASE  (0xFFFFF600)
#define _AT91C_PMC_BASE   (0xFFFFFC00)
#define _PIO_PER_OFFS     (0x00)
#define _PIO_OER_OFFS     (0x10)
#define _PIO_CODR_OFFS    (0x34)    /* Clear output data register   */
#define _PMC          (*(volatile unsigned int*) _AT91C_PMC_BASE)
#define _USB_ID       (_PIOB_ID)
#define _USB_OER
  (*(volatile unsigned int*) (_AT91C_PIOB_BASE + _PIO_OER_OFFS))
#define _USB_CODR
  (*(volatile unsigned int*) (_AT91C_PIOB_BASE + _PIO_CODR_OFFS))
#define _USB_DP_PUP_BIT (1)

static void _HWAttach(void) {
  _PMC      = (1 << _USB_ID);          /* Enable peripheral clock for USB-Port */
  _USB_OER  = (1 << _USB_DP_PUP_BIT);  /* set USB_DP_PUP to output */
```

 © 2010-2026 SEGGER Microcontroller GmbH

```
  _USB_CODR = (1 << _USB_DP_PUP_BIT);  /* set _USB_DP_PUP_BIT to low state */
}

static void _EnableISR(USB_ISR_HANDLER * pfISRHandler) {
  *(U32*)(0xFFFFF080 + 4 * PID_USB) = (U32)pfISRHandler; // Set interrupt vector
  *(U32*)(0xFFFFF128)               = (1 << PID_USB);
  // Clear pending interrupt
  *(U32*)(0xFFFFF120)               = (1 << PID_USB);    // Enable Interrupt
}

void USBD_X_Config(void) {
  USBD_AddDriver(&USB_Driver_AtmelSAM7A3);
  USBD_SetAttachFunc(_HWAttach);
  USBD_SetISREnableFunc(_EnableISR);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

26.2.2 USBD_X_DisableInterrupt()

Description

This function is called by the stack in cases where the stack must perform a critical operation which can not be interrupted by a new incoming USB interrupt event.

Prototype

```
void USBD_X_DisableInterrupt(void);
```

Additional information

This function is MCU/USB controller specific. Normally it is defined in the hardware specific `USB_Config_*.c` file.

This function is only called by the stack if the define `USBD_OS_USE_USBD_X_INTERRUPT` is set to 1 in `USB_Conf.h`. If this is not the case interrupts are disabled/enabled globally.

Example

```
/* Example excerpt from USB_Config_SEGGER_emPower.c */
void USBD_X_DisableInterrupt(void) {
  NVIC_DisableIRQ(USBHS_IRQn);
}

/* Example excerpt from USB_Config_Renesas_RSKRX71M.c */
void USBD_X_DisableInterrupt(void) {
  USB0_IER_USB0   &= ~(1uL << USB0_IER_USB0_BIT);
  USB0_IER_USBR0  &= ~(1uL << USB0_IER_USBR0_BIT);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

26.2.3 USBD_X_EnableInterrupt()

Description

This function is called by the stack to enable USB interrupt(s) after they have been disabled by `USBD_X_DisableInterrupt()`.

Prototype

```
void USBD_X_EnableInterrupt(void);
```

Additional information

This function is MCU/USB controller specific. Normally it is defined in the hardware specific `USB_Config_*.c` file.

This function is only called by the stack if the define `USBD_OS_USE_USBD_X_INTERRUPT` is set to 1 in `USB_Conf.h`. If this is not the case interrupts are disabled/enabled globally.

Example

```
/* Example excerpt from USB_Config_SEGGER_emPower.c */
void USBD_X_EnableInterrupt(void) {
  NVIC_EnableIRQ(USBHS_IRQn);
}

/* Example excerpt from USB_Config_Renesas_RSKRX71M.c */
void USBD_X_EnableInterrupt(void) {
  USB0_IER_USB0   |= (1uL << USB0_IER_USB0_BIT);
  USB0_IER_USBR0  |= (1uL << USB0_IER_USBR0_BIT);
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

26.3 Device driver specifics

For emUSB-Device different USB controller drivers are provided. Normally, the drivers are ready and do not need to be configured at all. Some drivers may need to be configured in a special manner, due to some limitation of the controller.

This section lists the drivers which require special configuration and describes how to configure those drivers.

Restrictions caused by the USB controller hardware are also listed in this section.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.1 LPC54/55xxx full-speed driver

This driver is used for the MCUs:

- LPC54608
- LPC540xx/54S0xx
- LPC55xxx
- LPC51U68

Configuration

This driver needs a memory area to store the endpoint table and endpoint transfer buffers. By default, the dedicated USB RAM attached to the high-speed controller is used. A different memory area can be assigned to the driver by calling the function `USBD_AssignMemory()` in `USBD_X_Config()`. This is required by some devices where the USB RAM is on a different memory address or if the full-speed and high-speed controllers are used simultaneously.

Minimum required memory: 256 bytes + 'maximum packet size' for each used non-control endpoint.

The memory area must be aligned to a 256-byte boundary.

26.3.2 LPC54/55xxx high-speed driver

This driver is used for the MCUs:

- LPC54608
- LPC540xx/54S0xx
- LPC55xxx
- iMXRT5xx

Restrictions

In some versions of the LPC54xxx MCUs, the high-speed device controller contains a serious bug: Under some circumstances the first byte of a data packet transferred to the host is changed to `0x00` (See LPC546xx errata sheet, Rev. 2.1, 23 October 2018, USB.15). Although the workaround suggested by NXP is implemented in the driver, data packets may still get corrupted if IN and OUT endpoints are active at the same time. This can't be avoided in many applications. This problem is known for the LPC54608 and some early samples of the LPC54018/LPC54S018. We recommend not using the high-speed device controller on devices with this issue. When in doubt, check with NXP for a specific device.

The high-speed device controller is not able to send ISO packets of size 1024 to the host. Only packet sizes up to 1023 bytes will work.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.3 EHCI driver

This driver is used for the MCUs:

- LPC18xx
- LPC43xx
- Kinetis (HS controller)
- iMX RT105x/RT106x/RT118x
- Zynq70xx

Configuration

This driver needs a memory area to store the DMA descriptors and endpoint transfer buffers. The memory must be provided by the application and must be passed to the USB stack using the function `USBD_AssignMemory()`.

Example

```
USBD_AddDriver(&USB_Driver_NXP_LPC43xx_DynMem);
USBD_AssignMemory(_MemPool, sizeof(_MemPool));
```

Minimum required memory (for systems without cached memory):

- 918 to 1536 bytes (depending on the number of endpoints the controller provides)
- For each used non-control OUT (RX) endpoint: 32 bytes + 'maximum packet size' of the endpoint
- For each used non-control IN (TX) endpoint: 64 bytes

Minimum required memory (for systems using cached memory):

- 1046 to 1664 bytes (depending on the number of endpoints the controller provides)
- For each used non-control endpoint (IN or OUT): 96 bytes + 2 * 'maximum packet size' of the endpoint

> **Warning**
>
> The memory area must be valid for DMA access by the USB controller.

The memory area should be aligned to a 2048-byte boundary to avoid wasting memory.

For some targets there also exists a variant of the driver that uses a memory area declared inside the driver code. When selecting this kind of driver (without the "`_DynMem`" suffix in the driver name), `USBD_AssignMemory()` must not be called. Instead, the size of the memory area can be configured by setting the preprocessor symbol `USB_ENDPOINT_BUFFER_POOL_SIZE` in `USB_Conf.h`.

Cache support

If the driver is installed on a system using cached (data) memory, cache functions for cleaning and invalidating cache lines must be provided and set with `USBD_SetCacheConfig()`.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.4 Synopsys DWC2 driver (slave mode)

This driver does not use DMA and is applicable for the MCUs:

- STM32F105/107
- STM32F2xx
- STM32F4xx
- STM32F7xx
- STM32H7xx
- STM32L4x5/4x6/4x7/4x9
- STM32U575/585
- XMC45xx
- EFM32GGxxx

Configuration

This driver needs a memory area for endpoint transfer buffers. By default, a memory area declared inside the driver code is used. The size of this area can be configured by setting the preprocessor symbol `USB_ENDPOINT_BUFFER_POOL_SIZE` in `USB_Conf.h`.

For the STM32F7xx and STM32H7xx drivers the memory may be provided by the application instead. In this case, the dynamic memory variant of the driver must be added to the USB stack and the function `USBD_AssignMemory()` must be called.

Example

```
USBD_AddDriver(&USB_Driver_ST_STM32F7xxHS_DynMem);
USBD_AssignMemory(_MemPool, sizeof(_MemPool));
```

Minimum required memory:

- 64 bytes
- For each used non-control OUT (RX) endpoint: The ’maximum packet size’ of the endpoint

Restrictions

High bandwidth ISO transactions are not supported; therefore, a maximum of 1024 bytes can be transferred per microframe in high-speed.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.5 Synopsys DWC2 driver (DMA mode)

This driver can be used for the high-speed controllers of the MCUs:

- STM32F4xx
- STM32F7xx
- STM32H7xx
- STM32U5A9
- DA148xx

Configuration

This driver needs a memory area for endpoint transfer buffers, which must be provided by the application by calling the function `USBD_AssignMemory()` after `USBD_AddDriver()`.

Minimum required memory:

- 128 bytes
- For each used non-control endpoint (IN and OUT each): The 'maximum packet size' of the endpoint

All sizes must be rounded up to a multiple of 4 bytes or the cache line size, if the system uses a cache.

> **Warning**
>
> The memory area must be valid for DMA access by the USB controller.

Cache support

If the driver is installed on a system using cached (data) memory, cache functions for cleaning and invalidating cache lines must be provided and set with `USBD_SetCacheConfig()`.

Some devices only use a data cache for specific memory areas (e.g., the STM32U5A9 for external memory). On such devices a driver version without cache management can be used, if the cached area is not used (or not used for USB).

Restrictions

For ISO OUT endpoints the maximum packet size must meet the following requirements:

- The maximum packet size should be a multiple of 4 bytes to avoid performance drawbacks.
- If the total packet size (number of bytes per micro frame) is > 1024 and ≤ 2048 (high bandwidth transactions), it MUST be a multiple of 8 bytes.
- If the total packet size (number of bytes per micro frame) is > 2048 and ≤ 3072 (high bandwidth transactions), it MUST be a multiple of 12 bytes.

Large packet sizes require large FIFO buffers which are usually not available in STM32 devices.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.6 XHCI driver

Configuration

The function `USBD_EnableSuperSpeed()` must be called within `USBD_X_Config()`, if the device shall be able to operate at SuperSpeed. If the function is not called, the USB controller will enumerate in high-speed only.

This driver needs a memory area for endpoint transfer buffers which must be provided by the application using the `USBD_AssignMemory()` function.

Typical required memory (on a system with a data cache line size of 64 bytes):

- 1152 bytes
- For each used non-control OUT (RX) endpoint: The ’maximum packet size’ of the endpoint + 512 bytes
- For each used non-control IN (TX) endpoint: 192 bytes

For optimal and reproducible memory allocation behavior, the memory area provided to `USBD_AssignMemory()` should be cache aligned and should not span a 64 KB boundary.

Cache support

Cache functions for cleaning and invalidating cache lines must be provided and set with `USBD_SetCacheConfig()`.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.7 Renesas RX driver

This driver is used for the MCUs:

- RX113
- RX231
- RX71M (full-speed controller)
- RX62N
- RX63N
- RX64M
- RX65N
- Synergy series
- RA4xx series

Restrictions

Due to a hardware limitation, the maximum packet size of isochronous endpoints is limited to 256 bytes (instead of the normally possible 1023 bytes).

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.8 AT91RM9200 driver

Restrictions

SETUP OUT transfers with more than 8 bytes can cause the controller to lock up. A setup OUT transfer consists of a SETUP stage, an optional DATA OUT stage and a STATUS IN stage. The usage of status IN with preceding data is relatively rare. Certain HID commands can trigger this, e.g., “SetFeature”. In most other protocols, setup transfers are rarely done in the OUT direction with a data stage. Usage of common protocols (MSD, CDC, etc.) should not be affected. The USB controller in this MCU appears to have a critical bug with status IN transactions which results in a complete lockup of the controller until a power cycle. The issue occurs during the status stage of setup transfers consisting of more than one data packet:

Works:

- SETUP packet
- 7-byte data OUT
- 0-byte status IN

Does not work:

- SETUP packet
- 8-byte data OUT
- 1-byte data OUT
- 0-byte status IN <-- During this transaction the IN token can be seen, and on the device side, the ZLP transaction is started, but the controller locks up. After this, the controller will no longer receive any interrupts.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.9 Giga Device GD32F4xx driver (full-speed controller)

The driver was tested on the GD32F450, GD32F470 and GD32F407.

Restrictions

Stress tests show that on the GD32F407 concurrent data transfers over multiple endpoints may result in data corruption. Therefore, using multiple independent USB classes in a device is not recommended for the GD32F407.

For ISO transactions the maximum packet size is 512 bytes.

26.3.10 Giga Device GD32F4xx driver (high-speed controller)

The driver was tested on the GD32F450, GD32F470 and GD32F407.

Configuration

The driver needs a memory area for endpoint transfer buffers, which must be provided by the application by calling the function `USBD_AssignMemory()` after `USBD_AddDriver()`.

Minimum required memory:

- 128 bytes
- For each used non-control endpoint (IN and OUT each): The ’maximum packet size’ of the endpoint

All sizes must be rounded up to a multiple of 4 bytes.

Restrictions

On the GD32F407, the USB controller doesn’t work reliably. Sporadic corruption of data packets may happen.

Stress tests show that on the GD32F450 concurrent data transfers over multiple endpoints may result in data corruption or may cause an endpoint to get stuck. Therefore, using multiple independent USB classes in a device is not recommended for the GD32F450.

ISO IN transactions only work with an interval of 125us.

For ISO transactions the maximum packet size is 800 bytes.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.11 Atmel ATSAMV7x driver

This driver can be used for MCUs:

- ATSAMV70
- ATSAMV71
- ATSAMV72
- ATSAME70

Restrictions

Due to a controller limitation, `USBD_Write*` functions can return before data is actually sent out. E.g., if the application calls `USBD_BULK_Write(hInst, pBuff, 50, 0)` and there is space in the endpoint's memory banks the function will immediately return 50 even if the host is not currently reading on the IN endpoint.

Normally this is not an issue, as the data can still be read normally by the host later on. But this can lead to confusion or can cause problems if a protocol is used which depends on knowing whether the host really received the packet.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.12 PSOC driver

This driver can be used for MCUs:

- Infineon PSOC 62
- Infineon PSOC Control C3

Two USB drivers are available in two versions: One is using DMA for data transfer; the other does not use DMA.

Driver objects for `USBD_AddDriver()`:

- `USB_HW_DRIVER USB_Driver_Cypress_PSoC6`
- `USB_HW_DRIVER USB_Driver_Cypress_PSoC6_DMA`
- `USB_HW_DRIVER USB_Driver_Infineon_PSC3M6`
- `USB_HW_DRIVER USB_Driver_Infineon_PSC3M6_DMA`

The DMA-capable driver needs a memory area for endpoint transfer buffers which must be provided by the application using the `USBD_AssignMemory()` function. The memory area should be word aligned. The required memory size can be calculated as follows:

- For each used non-control endpoint: The 'maximum packet size' of the endpoint, rounded up to a multiple of 32 bytes + 84 bytes

Additionally, the DMA routines and DMA channels must be configured using a call to `USB_DRIVER_Cypress_PSoC6_ConfigDMA()`.

26.3.12.1 Restrictions

Because the USB controller of the PSOC is not able to signal a suspend state of the USB bus via interrupt, the application has to call a driver function every millisecond in order to get suspend events handled, see `USB_DRIVER_…_SysTick()` functions.

For the non-DMA driver, ISO endpoints are limited to a maximum packet size of 192 bytes. Also, the sum of all endpoints' maximum packet sizes must be less than 512 bytes.

There are no such limits for the DMA driver.

26.3.12.2 PSOC driver specific functions

26.3.12.2.1 USB_DRIVER_Cypress_PSoC6_SysTick()

Description

The USB controller of the PSoC6 is not able to automatic detect a suspend issued by the host. In order to allow the driver to detect the suspend state, this function has to be called by the application every millisecond.

Calling this functions is also required when using LPM (`USB_PSOC6_LPM_CONFIG` ≠ 0).

If this function is not used, then USB stack works well, only suspend is not handled. This requires setting `USB_PSOC6_LPM_CONFIG` to 0.

Prototype

```
void USB_DRIVER_Cypress_PSoC6_SysTick(void);
```

26.3.12.2.2 USB_DRIVER_Cypress_PSoC6_Resume()

Description

If the device was set into deep sleep mode while USB was active, some registers of the USB controller are reset. This function can be called after the device has left deep sleep mode to restore the state of the USB controller.

Prototype

```
void USB_DRIVER_Cypress_PSoC6_Resume(void);
```

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.12.2.3 USB_DRIVER_Cypress_PSoC6_ConfigDMA()

Description

Configure DMA parameters and DMA routines to be used by the USB driver. This function must be called after `USBD_AddDriver()` when the DMA version of the PSoC6 USB driver was selected.

Prototype

```
void USB_DRIVER_Cypress_PSoC6_ConfigDMA(const USB_CYPRESS_PSoC6_DMA_API    * pAPI,
                                        const USB_CYPRESS_PSoC6_DMA_CONFIG * pCFG);
```

Parameters

Parameter	Description
pAPI	Pointer to a set of DMA driver functions to be used by the USB driver.
pCFG	Pointer to a structure that contains the DMA channel and priority for each USB endpoint.

26.3.12.2.4 USB_DRIVER_Infineon_PSC3M6_SysTick()

Description

The USB controller of the PSoC6 is not able to automatic detect a suspend issued by the host. In order to allow the driver to detect the suspend state, this function has to be called by the application every millisecond.

Calling this functions is also required when using LPM (`USB_PSC3M6_LPM_CONFIG` ≠ 0).

If this function is not used, then USB stack works well, only suspend is not handled. This requires setting `USB_PSC3M6_LPM_CONFIG` to 0.

Prototype

```
void USB_DRIVER_Infineon_PSC3M6_SysTick(void);
```

26.3.12.2.5 USB_DRIVER_Infineon_PSC3M6_Resume()

Description

If the device was set into deep sleep mode while USB was active, some registers of the USB controller are reset. This function can be called after the device has left deep sleep mode to restore the state of the USB controller.

Prototype

```
void USB_DRIVER_Infineon_PSC3M6_Resume(void);
```

26.3.12.2.6 USB_DRIVER_Infineon_PSC3M6_SetBaseAddr()

Description

Set the base address of the USB controller. This is used to switch between the secure and non-secure register set of the USB controller. Allowed values are: `0x52C20000` for the secure area, `0x42C20000` for the non-secure area (default).

Prototype

```
void USB_DRIVER_Infineon_PSC3M6_SetBaseAddr(U32 BaseAddr);
```

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.12.2.7 USB_DRIVER_Infineon_PSC3M6_ConfigDMA()

Description

Configure DMA parameters and DMA routines to be used by the USB driver. This function must be called after `USBD_AddDriver()` when the DMA version of the PSC3M6 USB driver was selected.

Prototype

```
void USB_DRIVER_Infineon_PSC3M6_ConfigDMA
                                    (const USB_Infineon_PSC3M6_DMA_API    * pAPI,
                                     const USB_Infineon_PSC3M6_DMA_CONFIG * pCFG);
```

Parameters

Parameter	Description
`pAPI`	Pointer to a set of DMA driver functions to be used by the USB driver.
`pCFG`	Pointer to a structure that contains the DMA channel and priority for each USB endpoint.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.13 ST full-speed driver

This driver can be used for MCUs:

- STM32H5xx
- STM32U545

This driver needs a memory area used as an endpoint transfer buffer which must be provided by the application using the `USBD_AssignMemory()` function.

The memory area must be word (32-bit) aligned, and its size must be the maximum packet size of the largest OUT (RX) endpoint used in the USB configuration. So usually 64 bytes are sufficient, except if any ISO OUT endpoint is used.

Restrictions

The maximum packet size of an interrupt endpoint must be even. The maximum packet size of an isochronous endpoint must be a multiple of 32 bytes and must not exceed 992 bytes.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.14 PSOCE84 driver

Configuration

This driver needs a memory area for endpoint transfer buffers, which must be provided by the application by calling the function `USBD_AssignMemory()` after `USBD_AddDriver()`.

Minimum required memory:

- 128 bytes
- For each used non-control endpoint (IN and OUT each): The 'maximum packet size' of the endpoint

All sizes must be rounded up to a multiple of 4 bytes or the cache line size, if the system uses a cache.

> **Warning**
>
> The memory area must be valid for DMA access by the USB controller.

Cache support

If the driver is running on the Cortex-M55 CPU, cache functions for cleaning and invalidating cache lines must be provided and set with `USBD_SetCacheConfig()`.

Restrictions

Due to a restricted hardware configuration the size of high bandwidth isochronous OUT packets is limited to a maximum of 2048 bytes (2 x 1024).

For ISO OUT endpoints the maximum packet size must meet the following requirements:

- The maximum packet size should be a multiple of 4 bytes to avoid performance drawbacks.
- If the total packet size (number of bytes per microframe) is > 1024 and ≤ 2048 (high bandwidth transactions), it MUST be a multiple of 8 bytes.

26.3.14.1 PSOCE84 driver-specific functions

26.3.14.1.1 USB_DRIVER_Infineon_PSOCE84_DMA_SetPLLRefClock()

Description

Set the PLL reference clock for the USB PHY. This function must be called in the `USB_X_Config()` function.

Prototype

```
void USB_DRIVER_Infineon_PSOCE84_DMA_SetPLLRefClock(unsigned MHz);
```

Parameters

Parameter	Description
`MHz`	Reference clock in `MHz`. Valid values are 9 (= 9.6 `MHz`), 10, 12, 19 (= 19.2 `MHz`), 20, 24 or 50.

 © 2010-2026 SEGGER Microcontroller GmbH

26.3.15 Atmel / Microchip high-speed driver

There are two versions of the driver: One using DMA and the other without DMA.

This driver can be used for MCUs:

- SAM3U (no DMA)
- SAM9X25 (no DMA)
- SAM9X75
- SAMA5D2x
- SAMA5D3x
- SAMA5D4x

Restrictions

ISO OUT data packets are not always transferred correctly. For some data patterns the USB controller hardware will corrupt the last three bytes of such packets.

The USB controller hardware silently discards zero-length OUT packets when using DMA without informing the software. This may be a problem for USB protocols that require explicit detection of zero-length packets. For example, the API function `USBD_BULK_Read-Record()` does not work if using the driver with DMA.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 27

Support

 © 2010-2026 SEGGER Microcontroller GmbH

27.1 Contacting support

Before contacting support, please make sure that you are using the latest version of the emUSB-Device package. Also, please check the chapter *Configuring debugging output* on page 49 and run your application with debug support enabled.

If you are a registered emUSB-Device user, there are different ways to contact the emUSB-Device support:

1. You can create a support ticket via email to ticket_emusb@segger.com*
2. You can create a support ticket at {segger.com/ticket}.

Please include the following information in the email or ticket:

- The emUSB-Device version.
- Your emUSB-Device license number.
- If you are unsure about the above information, you can also use the name of the emUSB-Device zip file (which contains the above information).
- A detailed description of the problem.
- The configuration files `USB_Conf*.*`.
- Any error messages.

Please also take a few moments to help us improve our services by providing short feedback once your support case has been solved.

27.1.1 Where can I find the license number?

The license number is part of the shipped zip file name.
For example, `emUSBD_BASE_STM32F2F4F7_V3.60.0_USBD-01234_308746BB_230530.zip` where USBD-01234 is the license number. The license number is also part of every *.c- and *.h-file header. For example, if you open USB.h you should find the license number as with the example below:

```
*********************************************************************
*                                                                   *
*       emUSB-Device version: V3.60.0                               *
*                                                                   *
*********************************************************************
---------------------------------------------------------------------
Licensing information
Licensor:                 SEGGER Microcontroller GmbH
Licensed to:              Customer name
Licensed SEGGER software: emUSB-Device
License number:           USBD-01234
License model:            SSL
Licensed product:         -
Licensed platform:        Cortex-M, GCC
Licensed number of seats: 1
---------------------------------------------------------------------
Support and Update Agreement (SUA)
SUA period:               2023-05-30 - 2023-11-30
Contact to extend SUA:    sales@segger.com
---------------------------------------------------------------------
Purpose : USB stack API
```

*By sending us an email your (personal) data will automatically be processed. For further information please refer to our privacy policy which is available at https://www.segger.com/legal/privacy-policy/.

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 28

Profiling with SystemView

This chapter describes how to configure and enable profiling of emUSB-Device using SystemView.

 © 2010-2026 SEGGER Microcontroller GmbH

28.1 Profiling overview

emUSB-Device is instrumented to generate profiling information of API functions and driver-level functions.

This profiling information exposes the run-time behavior of emUSB-Device in an application, recording which API functions have been called, how long the execution took, and revealing which driver-level functions have been called by API functions or events like interrupts.

The profiling information is recorded using SystemView.

SystemView is a real-time recording and visualization tool for profiling data. It exposes the true run-time behavior of a system, going far deeper than the insight provided by debuggers. This is particularly effective when developing and working with complex systems comprising an OS with multiple threads and interrupts, and one or more middleware components.

SystemView can ensure a system performs as designed, track down inefficiencies, and show unintended interactions and resource conflicts.

The recording of profiling information with SystemView is minimally intrusive to the system and can be done on virtually any system. With SEGGER's Real Time Technology (RTT) and a J-Link, SystemView can record data in real-time and analyze the data live, while the system is running.

The emUSB-Device profiling instrumentation can be easily configured and set up.

 © 2010-2026 SEGGER Microcontroller GmbH

28.2 Additional files for profiling

Additional files are required on the target and PC sides for full functionality of SystemView.

28.2.1 Additional files on target side

The SystemView module needs to be added to the application to enable profiling. If not already part of the project, download the sources from https://www.segger.com/systemview.html and add them to the project.

Also make sure that `USB_SYSVIEW.c` from the `/USB/` directory is included in the project.

28.2.2 Additional files on PC side

For fully functional and readable outputs in the SystemView PC application, a description file for the corresponding middleware is required. This description file extends the values sent from the target to fully readable text outputs.

While SystemView already comes with the most recent description files at the time the SystemView release was built, these files might not be the latest available. The latest SystemView description files can be found in the emUSB-Device shipment in the folder `/Shared/SystemView/Description/`. You can copy these files over to the `Description` folder that comes with the SystemView package.

The version at the end of the SystemView description file does not have to match the exact version of the middleware it is used with. They are valid from this version onwards until a description file for a newer version is required.

 © 2010-2026 SEGGER Microcontroller GmbH

28.3 Enable profiling

Profiling can be included or excluded at compile-time and enabled at run-time. When profiling is excluded, no additional overhead in performance or memory usage is generated. Even when profiling is enabled, the overhead is minimal, due to the efficient implementation of SystemView.

To include profiling, define USBD_SUPPORT_PROFILE as 1 in the emUSB-Device configuration (USB_Conf.h) or in the project preprocessor defines.

By default, profiling is included when the global define SUPPORT_PROFILE is set.

```
#if defined(SUPPORT_PROFILE) && (SUPPORT_PROFILE)
   #ifndef   USBD_SUPPORT_PROFILE
     #define USBD_SUPPORT_PROFILE           1
   #endif
#endif
```

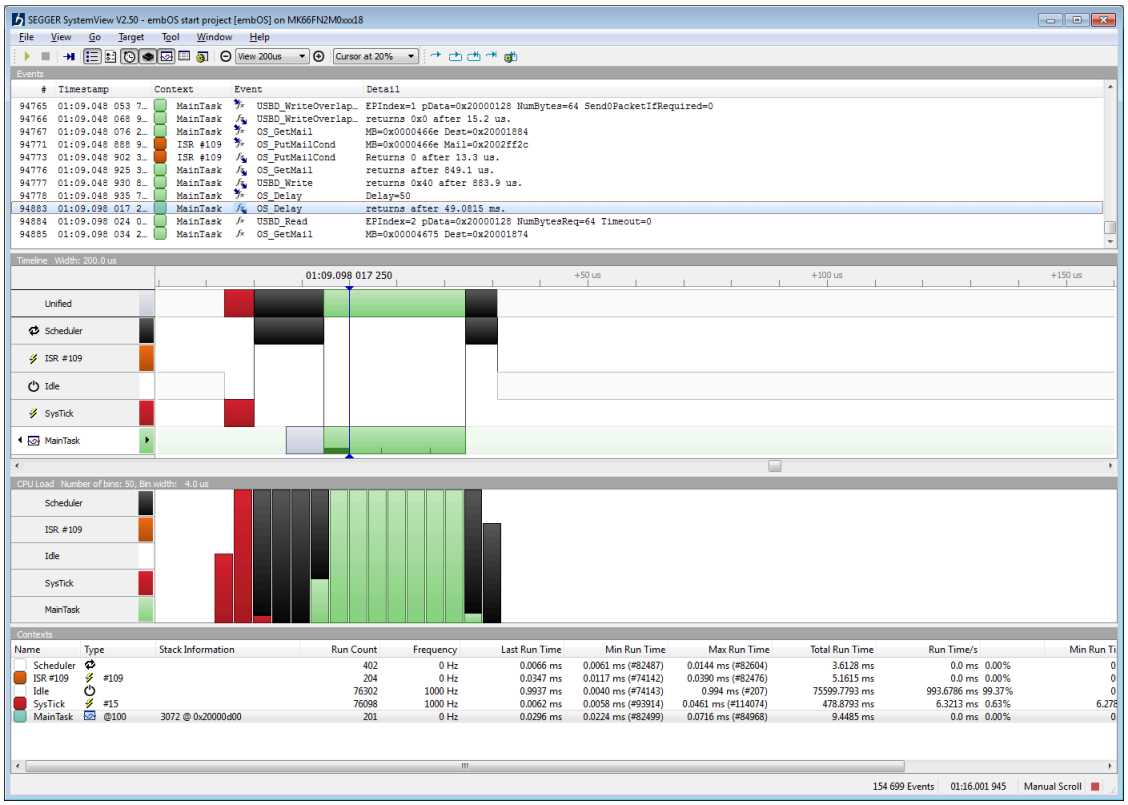
To enable profiling at run-time, USBD_SYSVIEW_Init() needs to be called. Profiling can be enabled at any time; it is recommended to do this in the user-provided configuration USBD_X_Config():

```
/*********************************************************************
*
*       USBD_X_Config
*
*/
void USBD_X_Config(void) {
   ...
#if USBD_SUPPORT_PROFILE
  USBD_SYSVIEW_Init();
#endif
   ...
```

 © 2010-2026 SEGGER Microcontroller GmbH

28.4 Recording and analyzing profiling information

When profiling is included and enabled, emUSB-Device generates profiling events. On a system which supports RTT (i.e., ARM Cortex-M and Renesas RX), the data can be read and analyzed with SystemView and a J-Link. Connect the J-Link to the target system using the default debug interface and start the SystemView host application. If the system does not support RTT, SystemView can be configured for single-shot or postmortem mode. Please refer to the SystemView User Manual for more information.



 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 29

Debugging

emUSB-Device comes with various debugging options. These include optional warning and log outputs, as well as other run-time options which perform checks at run time as well as options to drop incoming or outgoing packets to test stability of the implementation on the target system.

 © 2010-2026 SEGGER Microcontroller GmbH

29.1 Message output

The debug builds of emUSB-Device include a fine-grained debug system which helps to analyze the correct implementation of the stack in your application. All modules of the USB stack can output logging and warning messages via terminal I/O, if the specific message type identifier is added to the log and/or warn filter mask. This approach provides the opportunity to get and interpret only the logging and warning messages which are relevant for the part of the stack that you want to debug.

By default, all of the warning messages and none of the logging messages are activated. All activated messages are forwarded to the functions `USB_X_Log()` and `USB_X_Warn()`. These functions are located in the source file `USB_ConfigIO.c` and may be customized or replaced if necessary.

 © 2010-2026 SEGGER Microcontroller GmbH

29.2 API functions

Function	Description
Filter functions	
USBD_AddLogFilter()	Adds one or more message types to the logging filter.
USBD_AddWarnFilter()	Adds one or more message types to the warning filter.
USBD_SetLogFilter()	Sets the message type(s) for the logging filter.
USBD_SetWarnFilter()	Sets the message type(s) for the warning filter.
General debug functions/macros	
USB_PANIC	Called if the stack encounters a critical situation.
General helper prototypes	
USB_X_Log()	This function is called by the stack in debug builds with log output.
USB_X_Warn()	This function is called by the stack in debug builds with warning output.
USB_OS_Panic()	Is called if the stack encounters a fatal error.

 © 2010-2026 SEGGER Microcontroller GmbH

29.2.1 USBD_AddLogFilter()

Description

Adds one or more message types to the logging filter.

Prototype

```
void USBD_AddLogFilter(U32 FilterMask);
```

Parameters

Parameter	Description
FilterMask	Specifies which logging messages should be added to the filter mask. Refer to *Message types* on page 785 for a list of valid values for parameter FilterMask.

Additional information

USBD_AddLogFilter() can also be used to remove a filter condition which was set before. It adds the specified filter to the filter mask via a disjunction.

Example

```
void Application (void) {
  USBD_AddLogFilter(USB_MTYPE_DRIVER); // Activate driver logging messages
  USBD_Init();
  /*
   * Do something
   */
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

29.2.2 USBD_AddWarnFilter()

Description

Adds one or more message types to the warning filter.

Prototype

```
void USBD_AddWarnFilter(U32 FilterMask);
```

Parameters

Parameter	Description
FilterMask	Specifies which warning messages should be added to the filter mask. Refer to *Message types* on page 785 for a list of valid values for parameter FilterMask.

Additional information

USBD_AddWarnFilter() can also be used to remove a filter condition which was set before. It adds the specified filter to the filter mask via a disjunction.

Example

```
void Application (void) {
  USBD_AddWarnFilter(USB_MTYPE_DRIVER); // Activate driver warning messages
  USBD_Init();
  /*
   * Do something
   */
}
```

 © 2010-2026 SEGGER Microcontroller GmbH

29.2.3 USBD_SetLogFilter()

Description

Sets the message type(s) for the logging filter.

Prototype

```
void USBD_SetLogFilter(U32 FilterMask);
```

Parameters

Parameter	Description
FilterMask	Specifies which logging messages should be set to the filter mask. Refer to *Message types* on page 785 for a list of valid values for parameter FilterMask.

Additional information

This function can be called before USBD_Init(). By default, none of filter conditions are set. The sample application contain a simple implementation which can be easily modified.

 © 2010-2026 SEGGER Microcontroller GmbH

29.2.4 USBD_SetWarnFilter()

Description

Sets the message type(s) for the warning filter.

Prototype

```
void USBD_SetWarnFilter(U32 FilterMask);
```

Parameters

Parameter	Description
FilterMask	Specifies which warning messages should be set to the filter mask. Refer to *Message types* on page 785 for a list of valid values for parameter FilterMask.

Additional information

This function can be called before USBD_Init(). By default, none of filter conditions are set. The sample application contain a simple implementation which can be easily modified.

 © 2010-2026 SEGGER Microcontroller GmbH

29.2.5 USB_PANIC

Description

This macro is called by the stack code when it detects a situation that should not be occurring and the stack cannot continue. The intention of the `USB_PANIC()` macro is to invoke whatever debugger may be in use by the programmer. In this way, it acts like an embedded breakpoint. This macro is mostly used in cases where emUSB-Device was configured improperly.

Prototype

```
USB_PANIC (const char * sError);
```

Additional information

This macro maps to a function in debug builds only. If `USB_DEBUG` > 0, the macro maps to the stack's internal function `USB_OS_Panic()`. It disables all interrupts to avoid further task switches, outputs sError via terminal I/O and loops forever. When using an emulator, you should set a breakpoint at the beginning of this routine or simply stop the program after a failure. The error message is passed to the function as a parameter. In a release build, this macro is defined empty, so that no additional code will be included by the linker.

 © 2010-2026 SEGGER Microcontroller GmbH

29.2.6 USB_X_Log()

Description

This function is called by the stack in debug builds with log output. In a release build, this function is not be linked in.

Prototype

```
void USB_X_Log(const char * s);
```

Parameters

Parameter	Description
s	Pointer to a string holding the log message.

 © 2010-2026 SEGGER Microcontroller GmbH

29.2.7 USB_X_Warn()

Description

This function is called by the stack in debug builds with warning output. In a release build, this function is not be linked in.

Prototype

```
void USB_X_Warn(const char * s);
```

Parameters

Parameter	Description
s	Pointer to a string holding the warning message.

 © 2010-2026 SEGGER Microcontroller GmbH

29.2.8 USB_OS_Panic()

Description

Is called if the stack encounters a fatal error.

Prototype

```
void USB_OS_Panic(const char * pErrMsg);
```

Parameters

Parameter	Description
pErrMsg	Pointer to a string holding the error message.

Additional information

In a release build this function is not linked in. The default implementation of this function disables all interrupts to avoid further task switches, outputs the error string via terminal I/O and loops forever. When using an emulator, you should set a break-point at the beginning of this routine or simply stop the program after a failure.

 © 2010-2026 SEGGER Microcontroller GmbH

29.3 Message types

Description

The same message types are used for log and warning messages. Separate filters can be used for both log and warnings. For details, refer to `USBD_SetLogFilter()` and `USBD_SetWarnFilter()` as wells as `USBD_AddLogFilter()` and `USBD_AddWarnFilter()` for more information about using the message types.

Definition

```
#define USB_MTYPE_INIT           (1UL <<  0)
#define USB_MTYPE_CORE           (1UL <<  1)
#define USB_MTYPE_CONFIG         (1UL <<  2)
#define USB_MTYPE_DRIVER         (1UL <<  3)
#define USB_MTYPE_ENUMERATION    (1UL <<  4)
#define USB_MTYPE_CDC            (1UL <<  7)
#define USB_MTYPE_HID            (1UL <<  8)
#define USB_MTYPE_MSD            (1UL <<  9)
#define USB_MTYPE_MSD_CDROM      (1UL << 10)
#define USB_MTYPE_MSD_PHY        (1UL << 11)
#define USB_MTYPE_MTP            (1UL << 12)
#define USB_MTYPE_PRINTER        (1UL << 13)
#define USB_MTYPE_RNDIS          (1UL << 14)
#define USB_MTYPE_VIRTUAL_MSD    (1UL << 16)
#define USB_MTYPE_UVC            (1UL << 17)
#define USB_MTYPE_ECM            (1UL << 18)
#define USB_MTYPE_AUDIO          (1UL << 19)
#define USB_MTYPE_NCM            (1UL << 20)
#define USB_MTYPE_MIDI           (1UL << 21)
#define USB_MTYPE_INFO           (1UL << 31)
```

Symbols

Definition	Description
USB_MTYPE_INIT	Activates output of messages from the initialization of the stack that should be logged.
USB_MTYPE_CORE	Activates output of messages from the core of the stack that should be logged.
USB_MTYPE_CONFIG	Activates output of messages from the configuration of the stack.
USB_MTYPE_DRIVER	Activates output of messages from the driver that should be logged.
USB_MTYPE_ENUMERATION	Activates output of messages from enumeration that should be logged. Note: Since enumeration is handled in an ISR, use this with care as the timing will be changed greatly.
USB_MTYPE_CDC	Activates output of messages from CDC module that should be logged when a CDC connection is used.
USB_MTYPE_HID	Activates output of messages from HID module that should be logged when a HID connection is used.
USB_MTYPE_MSD	Activates output of messages from MSD module that should be logged when a MSD connection is used.
USB_MTYPE_MSD_CDROM	Activates output of messages from MSD CDROM module that should be logged.
USB_MTYPE_MSD_PHY	Activates output of messages from MSD Physical layer that should be logged.
USB_MTYPE_MTP	Activates output of messages from MTP module that should be logged when a MTP connection is used.

 © 2010-2026 SEGGER Microcontroller GmbH

Definition	Description
USB_MTYPE_PRINTER	Activates output of messages from Printer module that should be logged when Printer connection is used.
USB_MTYPE_RNDIS	Activates output of messages from RNDIS module that should be logged when a RNDIS connection is used.
USB_MTYPE_VIRTUAL_MSD	Activates output of messages from VirtualMSD module that should be logged when a VirtualMSD connection is used.
USB_MTYPE_UVC	Activates output of messages from UVC module that should be logged when a UVC connection is used.
USB_MTYPE_ECM	Activates output of messages from ECM module that should be logged when a ECM connection is used.
USB_MTYPE_AUDIO	Activates output of messages from Audio module that should be logged when an audio connection is used.
USB_MTYPE_NCM	Activates output of messages from NCM module that should be logged when a NCM connection is used.
USB_MTYPE_MIDI	Activates output of messages from MIDI module that should be logged when a MIDI connection is used.
USB_MTYPE_INFO	Non-maskable info messages

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 30

Performance & resource usage

This chapter covers the performance and resource usage of emUSB-Device. It contains information about the memory requirements in typical systems which can be used to obtain sufficient estimates for most target systems.

 © 2010-2026 SEGGER Microcontroller GmbH

30.1 Memory footprint

emUSB-Device is designed to fit many kinds of embedded design requirements. Several features can be excluded from a build to get a minimal system. The code size depends on the API functions called by the application. The code was compiled for a Cortex-M4 CPU with the SEGGER compiler and size optimization. Note that the values are only valid for an average configuration.

The following table shows the approximate RAM and ROM requirement of emUSB-Device in bytes:

Component	ISO	ROM	RAM	Note
USB core	no	5400	1000	
USB core	yes	5600	1000	
Bulk	no	2000	200	
CDC	no	1200	100	
HID	no	1600	200	
MSD	no	4900	500	+ size of file system + configurable sector buffer of minimum 512 bytes (RAM)
MTP	no	15900	1500	+ size of file system + configurable file data buffer of minimum 512 bytes (RAM) + configurable object buffer (typically 4 kBytes RAM)
Printer	no	900	2100	
RNDIS	no	5700	1600	+ size of the IP stack
ECM	no	3000	300	+ size of the IP stack
NCM	no	3500	2900	+ size of the IP stack
IP-Over-USB	no	7800	1700	+ size of the IP stack
VirtualMSD	no	8300	1000	+ heap of minimum 1700 bytes RAM
DFU	no	900	0	
AUDIO	yes	3600	200	+ static configuration data
MIDI	no	1700	0	
CCID	no	1100	0	
Driver Atmel SAM3U	no	2000	600	
Driver Atmel SAM3U	yes	2000	1100	
Driver Atmel SAM3X	no	1900	500	
Driver Atmel SAM3S	no	2100	100	
Driver Atmel SAM7S	no	2100	100	
Driver Atmel SAM9X25	no	1900	600	
Driver Atmel SAM9X25	yes	1900	1100	
Driver Atmel SAMA5D2x	no	2200	600	
Driver Atmel SAMA5D2x	yes	2200	1200	
Driver Atmel SAMV7	no	1700	600	
Driver EM EFM32GG990	no	2900	700	
Driver EM EFM32GG990	yes	3400	1800	
Driver Freescale KHCI	no	2100	400	

 © 2010-2026 SEGGER Microcontroller GmbH

Component	ISO	ROM	RAM	Note
Driver Freescale KinetisEHCI	no	2700	2600	
Driver Freescale KinetisEHCI	yes	2800	3700	
Driver Infineon XMC45xx	no	2900	700	
Driver Infineon XMC45xx	yes	3400	1800	
Driver NXP LPC17xx	no	1600	100	
Driver NXP LPC18xx	no	2700	4200	
Driver NXP LPC18xx	yes	2800	5200	
Driver NXP LPC23xx	no	1400	100	
Driver NXP LPC43xx	no	2700	4200	
Driver NXP LPC43xx	yes	2800	5200	
Driver Renesas RZ	no	2500	7900	
Driver Renesas RZ	yes	2700	7900	
Driver Renesas RX	no	2300	700	
Driver Renesas RX	yes	2400	700	
Driver Renesas SynergyS1	no	2200	600	
Driver Renesas SynergyS1	yes	2300	600	
Driver Renesas SynergyFS	no	2300	700	
Driver Renesas SynergyFS	yes	2500	700	
Driver Renesas SynergyHS	no	2600	4800	
Driver Renesas SynergyHS	yes	2700	4800	
Driver ST STM32x32	no	1600	300	
Driver ST STM32x32	yes	1900	1200	
Driver ST STM32F107	no	2900	400	
Driver ST STM32F107	yes	3400	1500	
Driver ST STM32F4xxFS	no	2900	400	
Driver ST STM32F4xxFS	yes	3400	1500	
Driver ST STM32F4xxHS	no	3100	2900	
Driver ST STM32F4xxHS	yes	3500	3900	
Driver ST STM32F7xxFS Dyn-Mem	no	3100	200	+ endpoint buffer RAM
Driver ST STM32F7xxFS Dyn-Mem	yes	3600	300	+ endpoint buffer RAM
Driver ST STM32F7xxHS DMA	no	3100	300	+ endpoint buffer RAM
Driver ST STM32F7xxHS DMA	yes	3800	400	+ endpoint buffer RAM
Driver ST STM32L4xx	no	3000	600	
Driver ST STM32L4xx	yes	3500	1700	
Driver ST STR91x	no	1300	0	
Driver TI AM335x	no	1300	500	
Driver TI OMAP L138	no	1400	500	
Driver TI OMAP L138	yes	1400	500	
Driver Xilinx Ultrascale0	no	3700	200	+ endpoint buffer RAM
Driver Xilinx Ultrascale0	yes	4700	300	+ endpoint buffer RAM

 © 2010-2026 SEGGER Microcontroller GmbH

Additionally, 64 or 512 bytes of RAM (64 for full-speed and 512 for high-speed devices) are necessary for each OUT-endpoint as a data buffer. This buffer is assigned within the application.

 © 2010-2026 SEGGER Microcontroller GmbH

30.2 Performance

The tests were run on an LPC4357 CPU running at 180 MHz using the USB Bulk component connected to a Linux host.

The following table shows the transfer speed of emUSB-Device:

Description	Speed
USB High-Speed controller (device to host)	44.1 MB/s
USB High-Speed controller (host to device)	41.8 MB/s
USB Full-Speed controller	1200 kB/s

 © 2010-2026 SEGGER Microcontroller GmbH

Chapter 31

FAQ

This chapter answers some frequently asked questions.

 © 2010-2026 SEGGER Microcontroller GmbH

Q: **When designing my hardware can I just permanently connect the D+ 1.5 kOhm pull-up resistor to 3.3V to save an MCU pin?**

A: No, the pull-up being connected tells the USB host that the device is ready to communicate. Permanently connecting the pull-up is dangerous as the host may start to communicate with the device prematurely, before it has finished with the start-up. Furthermore, when using USB high-speed the device must disconnect the pull-up from D + according to the USB 2.0 specification. Any currently known USB high-speed controller (with internal or external USB high-speed PHY) has internal logic to enable an internal pull-up initially. So for those controllers an external pull-up is not necessary.

Q: **When using MSD can I read/write onto the storage medium when the device is connected to a USB host?**

A: No, when an MSC device is connected to a USB host the host is the sole master of the storage medium. It can write or read at any point in time. Should the application try to access the storage medium at the same time as the host the results are unpredictable. To resolve this issue the device needs to detach the storage medium from the host, see `USBD_MSD_RequestDisconnect()`, `USBD_MSD_Disconnect()` and `USBD_MSD_WaitForDisconnection()`.

Q: **Can I combine different USB components together?**

A: Yes. See *Combining USB components (Multi-Interface)* on page 717.

Q: **Do I need a real-time operating system (RTOS) to use the emUSB-Device-MSD?**

A: No, if your target application is a pure storage application. You do not need an RTOS if all you want to do is to run emUSB-Device-MSD as the only task on the target device. If your target application is more than just a storage device and needs to perform other tasks simultaneously, you need an RTOS which handles the multitasking. We recommend using our embOS Real-time OS, since all example and trial projects are based on it.

Q: **Do I need extra file system code to use the emUSB-Device-MSD?**

A: No, if you access the target data only from the host. Yes, if you want to access the target data from within the target itself. There is no extra file system code needed if you only want to access the data on the target from the host side. The host OS already provides several file systems. You have to provide file system program code on the target only if you want to access the data from within the target application itself.

 © 2010-2026 SEGGER Microcontroller GmbH